UNIVERSITÄT
HEIDELBERG
ZUKUNFT
SEIT 1386

HOCHSCHULE HEILBRONN
TECHNIK WIRTSCHAFT INFORMATIK

Softwareplagiatserkennung auf Java-Bytecodebasis

Bachelor-Thesis

zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.) im Studiengang Medizinische Informatik

vorgelegt von
Maximus Mutschler
21. Juli 2014, Heilbronn

Referent: Prof. Dr. Daniel Pfeifer
Korreferent: Prof. Dr. Martin Haag
Betreuer: Dipl.-Inform. Med. Hauke Hund

Danksagung

Ich bedanke mich bei Herrn Prof. Dr. Pfeifer für seine tatkräftige Unterstützung und für all die Antworten, die er mir bereitwillig auf meine Fragen gegeben hat.

Großen Dank gilt auch meinem Betreuer Hauke Hund, der für jedes technische Problem eine Lösung parat hatte. Besonders will ich mich bei ihm für die Hilfe bei der Beschaffung und Auswertung der Evaluierungsdaten bedanken.

Weiterer Dank gebührt Fabian Sailer, Janek Gröhl und Richard Zowalla für ihre Unterstützung beim Erstellen von Plagiaten.

Zusammenfassung

Im Rahmen dieser Arbeit wird eine eigens entwickelte Plagiatserkennungsoftware für Java-Programme namens *Plagiarism Finder* vorgestellt. Der Plagiatserkennungsprozess dieser Software basiert auf dem Java-Bytecode. Es werden die Grundlagen der Plagiatserkennung und des Java-Bytecodes umfassend erläutert. Des Weiteren wird die Funktionsweise, der Entwurf, die Benutzeroberfläche und die Evaluierung von *Plagiarism Finder* dargestellt. Hierbei wird auf folgende von der Wissenschaft bisher nicht betrachteten Aspekte eingegangen:

- Die Art der Normalisierung des Bytecodes vor dem Vergleich.
- Wie gelöst werden kann, dass das Verschieben von Methoden keinen Einfluss auf die ermittelten Ergebnisse hat.
- Wie bei der Plagiatserkennung Vorlagen gefiltert werden können.

Die Arbeit kommt zu dem Resultat, dass sich die Ergebnisse von *Plagiarism Finder* stabil gegenüber Änderungen des Wortlautes, des Textsatzes und dem Verschieben von Methoden verhalten. Änderungen an Kontrollstrukturen (z.B. For- statt While-Schleifen), an Zugriffsmodifikatoren und an der Anzahl von Methoden führen zu einem instabilen Verhalten der Ergebnisse.
Alles in allem kann *Plagiarism Finder* mit einer etablierten Plagiatserkennugssoftware wie *JPlag* [MP00] mithalten. Auf Grundlage der untersuchten Daten ist *Plagiarism Finder* im Erkennen von Plagiaten leicht schlechter als *JPlag*. *Plagiarism Finder* ist hingegen deutlich besser im Erkennen von nicht-plagiierter Software. Bei wortwörtlichen Kopien sowie bei Änderungen an der Textgestaltung und an Bezeichnungen sind die Ergebnisse der Programme nahezu identisch.

Inhaltsverzeichnis

Abkürzungsverzeichnis

ACS Attribute-Counting System

BCEL Byte Code Engineering Library

GST Greedy String Tiling Algorithmus

JVM Java Virtual Machine

LCS Longest Common Subsequenz

PF Plagiarism Finder

TTL Total Tile Length

MML Minimum Match Length

SMS Structure Metrics System

BDSG Bundesdatenschutzgesetz

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einführung

1.1 Gegenstand und Motivation

Im Studiengang Medizinische Informatik an der Hochschule Heilbronn und der Universität Heidelberg hat sich bezüglich der Überprüfung von studentischen Abgaben auf Plagiate kein Programm zur Softwareplagiatserkennung etabliert. Ohne ein solches Programm ist die Suche nach Plagiaten in studentischen Abgaben zeitaufwändig. Darüber hinaus stehen die Korrektoren meist unter zeitlichem Druck und richten ihr Augenmerk mehr auf die Richtigkeit der Lösung, als auf etwaige Plagiate. Zudem werden von Studierenden erstellte Programme auf mehrere Korrektoren aufgeteilt. Dies führt dazu, dass Plagiate oft nicht erkannt werden, da diese und die ihnen entsprechenden Originale von verschiedenen Korrektoren überprüft werden.
Aus einem wissenschaftlichen Antrieb heraus ist die Frage aufgestellt worden, ob Plagiate von Java-Programmen durch einen Java-Bytecodevergleich effektiv gefunden werden können. Dieser Ansatz ist bereits von Beka und Manber [BU98] getestet und validiert worden. Allerdings sind von ihnen nicht mehr aktuelle Algorithmen und Programme verwendet worden, weshalb diese Thema in dieser Arbeit neu aufgegriffen wird.
Plagiatserkennung auf Basis eines Bytecodevergleichs ergibt die Möglichkeit, dass auch in Programmen, welche nur in kompilierter Form ohne Quellcode vorhanden sind, nach Plagiaten gesucht werden kann. Solche Programme kommen meist in einem wirtschaftlichen Kontext vor, in welchem zum Schutz des Eigenen Vorteils ein reges Interesse darin besteht, Plagiate zu finden.

Aus den genannten Gründen wurde im Kontext dieser Arbeit ein Programm entwickelt, welches auf Grundlage eines Java-Bytecodevergleichs Plagiate von Java-Programmen erkennt. Des Programm trägt den Namen *Plagiarism Finder (PF)*. *PF* soll zeigen, dass Plagiate durch Java-Bytecodevergleich effektiv gefunden werden können. Zusätzlich soll es das Finden von Plagiaten in studentischen Abgaben erheblich vereinfachen.

1.2 Zielsetzung

Mit dieser Bachelorarbeit soll folgendes erreicht werden:

1. Es soll ein Programm zur Softwareplagiatserkennung von Java-Programmen erstellt werden. Das Programm soll effektiv in der Erkennung von Plagiaten sein. Es sollen aktuelle Algorithmen implementiert werden und auf schon vorhandenen Programmierschnittstellen aufgebaut werden. Zusätzlich soll der Ähnlichkeitsvergleich zwischen Programmen auf Basis des Java-Bytecodes durchgeführt werden.

2. Das entwickelte Programm soll evaluiert werden. Um zu überprüfen, ob der Ansatz mit dem Java-Bytecode erfolgversprechend ist, soll die Güte des Programms anhand Beispieldaten getestet werden.
   Um die Praxistauglichkeit des Programms zu beweisen, sollen studentische Abgaben auf Plagiate untersucht werden. Zusätzlich soll ein Vergleich mit einem anderen Plagiatserkennungsprogramm, welches nicht auf einem Java-Bytecodevergleich basiert, durchgeführt werden.

1.3 Überblick

Die folgende Liste zeigt einen Überblick aller Kapitel.

Kapitel 2 Grundlagen:
In diesem Kapitel wird auf die notwendigen Grundlagen, welche für die Entwicklung und das Verständnis von *PF* notwendig sind, eingegangen.

Kapitel 3 Entwurf und Implementierung:
In diesem Kapitel wird *PF* vorgestellt. Es wird auf die Funktionsweise, die wesentlichen Klassen und die Funktionen der Benutzeroberfläche des Plagiatserkennungsprogramms eingegangen.

Kapitel 4 Evaluierung:
Dieses Kapitel beinhaltet mehrere Experimente, welche zur Evaluation von *PF* dienen.

Kapitel 5 Diskussion:
In der Diskussion werden die Ergebnisse der Experimente interpretiert und diskutiert. Weiterhin wird auf Stärken und Schwächen von *PF* eingegangen. Zuletzt wird ein Ausblick über weiterführende Entwicklungs- und Forschungsaspekte gegeben.

2 Grundlagen

Dieses Kapitel beinhaltet die Grundlagen, welche für das Verständnis des Entwurfs und der Evaluierung von *PF* notwendig sind. Im Abschnitt 2.1 wird auf die Grundlage der Softwareplagiatserkennung und ihre verschiedenen Ansätze eingegangen. Der Abschnitt 2.2 stellt den *Greedy String Tiling Algorithmus (GST)* vor. Dabei werden die Anforderungen an diesen, die aus ihm ableitbaren Ähnlichkeitswerte sowie ein Ansatz zur Filterung von Vorlagen erläutert. Im Abschnitt 2.4 werden die wesentlichen Elemente des Java-Bytecodes veranschaulicht. Abschnitt 2.5 gibt eine kurze Einführung in die *Byte Code Engineering Library (BCEL)*. Zuletzt wird im Abschnitt 2.6 darauf eingegangen, wie Plagiate eingestuft werden können.

2.1 Softwareplagiatserkennung

Dieser Abschnitt gibt einen Überblick über die Softwareplagiatserkennung und die verbreitetsten Ansätze um diese umzusetzen.

Von einem **Plagiat** spricht man, wenn es zu einem Diebstahl geistigen Eigentums durch unrechtmäßiges Nachahmen eines von einem anderen geschaffenen künstlerischen oder wissenschaftlichen Werkes kommt [17, S. 1045]. **„Softwareplagiat“** ist eine Spezialisierung des Begriffs „Plagiat“ und beschreibt das unrechtmäßige Übernehmen von Ideen und Quellcode aus einem Original.
Es gibt zwei Arten, ein Plagiat zu erstellen. Die erste hat ein **wörtliches Plagiat** zum Ergebnis [ASA12, S. 134]. Hierbei werden Teile eines Originals wortwörtlich übernommen. Um ein wörtliches Plagiat zu erkennen, müssen die Zeichen eines Plagiats mit denen eines möglichen Originals verglichen werden.
Die zweite Art eines Plagiates wird durch das Übernehmen und Abändern von Teilen eines Originals erreicht. Vom Plagiat kann dann nur durch die beschriebene Idee aber nicht durch den Wortlaut auf das Original zurückgeschlossen werden. Das Ergebnis dieser Art des Plagiierens nennt man **intelligentes Plagiat** [ASA12, S. 134]. Ein intelligentes Plagiat ist schwerer zu erkennen als ein wörtliches Plagiat, da in diesem Fall die Bedeutung der Worte, die Semantik, verglichen werden muss. Gegenüber dem Erkennen intelligenter Plagiate von Texten in natürlicher Sprache ist das Erkennen von intelligenten Softwareplagiaten im allgemeinen einfacher. Dies beruht auf der Tatsache, dass die Grammatik und somit die möglichen Strukturen einer Programmiersprache

sowie die Bedeutung dieser Strukturen klar definiert sind. Bei einer natürlichen Sprache ist dies nicht der Fall [Clo00, S. 7]. Dass Softwareplagiate leichter zu entdecken sind, gilt demnach aber nur, wenn man sich bei der Suche nach Plagiaten auf den Vergleich von Strukturen der Programmiersprache beschränkt und natürlichsprachliche Elemente, wie zum Beispiel Bezeichnungen, ignoriert.
Zur Erkennung von Softwareplagiaten gibt es mehrere Ansätze. Zu den am weitesten Verbreiteten gehören:

Attribute-Counting System (*ACS*)
Ein solches System bestimmt durch die Häufigkeit des Auftretens bestimmter Eigenschaften Kenngrößen, welche die Ähnlichkeit zweier Programme beschreiben [WV96, S 1].

Structure Metrics System (*SMS*)
Ein *SMS* bestimmt die Ähnlichkeit zweier Programme durch Vergleich der Struktur der Programme. Hierfür werden meistens Zeichenrepräsentationen der Struktur von Programmen erstellt und diese auf übereinstimmende Passagen verglichen. Eine solche Zeichenrepräsentation ist eine einfache Zeichenkette. Aus dem Vergleich der Zeichenrepräsentationen zweier Programme werden Maße bestimmt, welche die Ähnlichkeit der Programme beschreiben [WV96, S. 2].

Ein Beispiel für ein *ACS* ist die Bestimmung der Ähnlichkeit über die *Halstead-Metrik* [Hal77]. Dabei sind die einzelnen zu vergleichenden Eigenschaften:

- η_1: Menge der unterschiedlichen Operatoren des ersten Programms
- η_2: Menge der unterschiedlichen Operatoren des zweiten Programms
- N_1: Gesamtmenge der Operatoren des ersten Programms
- N_2: Gesamtmenge der Operatoren des zweiten Programms

Ottenstein [Ott76] setzte diesen Ansatz in einem der ersten automatischen Plagiatserkennungssysteme ein. Er nimmt dabei an, dass die Werte η_1, η_2, N_1 und N_2 bei Plagiaten ähnlich sind.

Ein *SMS* lässt sich wie folgt beispielhaft darstellen:
Eine Form der Zeichenrepräsentation der Struktur eines Programms stellt der Quelltext selbst dar. Die Ähnlichkeit zweier Zeichenströme kann über die *Longest Common Subsequenz (LCS)* bestimmt werden. Die *LCS* ist die größte Sequenz an Zeichen, welche in zwei Zeichenketten identisch ist. Hinsichtlich eines Plagiats bedeutet dies, dass, umso länger die *LCS* ist, desto größer sind die zusammenhängenden Übereinstimmungen zweier Programme. Die *LCS* wird als Wert zur Bestimmung der Ähnlichkeit in der Plagiatserkennungssoftware YAP eingesetzt [Wis96, S. 1].

Beim Vergleich der beiden Verfahren kommen Verco und Wise [WV96] zu dem Ergebnis, dass ein *ACS* bei unerfahrenen Plagiatoren erfolgreicher sein kann als ein *SMS*, allerdings im Großen und Ganzen ein *SMS* besser dazu geeignet ist, Plagiate zu erkennen. Deshalb baut die in dieser Arbeit vorgestellte Plagiatserkennungssoftware auf einem *SMS*-Ansatz auf.

2.2 Greedy String Tiling Algorithmus

Dieser Abschnitt beinhaltet eine Beschreibung des *Greedy String Tiling Algorithmus (GST)*. Dazu gehört eine Beschreibung der Anforderungen an den *GST* in Abschnitt 2.2.1, Begriffe zum Verständnis in Abschnitt 2.2.2, die Erklärung der Funktionsweise in Abschnitt 2.2.3 sowie die Darstellung der aus ihm ableitbaren Ähnlichkeitsmaße in Abschnitt 2.2.4. Ein Verständnis des *GST* ist notwendig, da er sowie die über ihn ermittelten Ähnlichkeitsmaße wesentliche Bestandteile von *PF* sind.
Allgemein dient der *GST* zur Erkennung exakter Übereinstimmungen zwischen zwei Zeichenketten. Er wird in *SMSs* wie zum Beispiel YAP3 [Wis96] eingesetzt. Die nun folgenden Abschnitte 2.2.1, 2.2.2 und 2.2.3 sind zusätzlich eigener Ergänzungen inhaltlich an die Arbeit von Wise [Wis94, S 1-4] angelehnt. Die Formulierungen „Substring“ und „Teil einer Zeichenkette“ werden im Folgenden als Synonym verwendet.

2.2.1 Anforderungen an die Ähnlichkeitsbestimmung

Für die Bestimmung der Ähnlichkeit zwischen zwei Zeichenketten gelten die folgenden drei Anforderungen:

Anforderung 1: Eindeutigkeit
Jeder Teil einer Zeichenkette darf nur einmal, falls eine Übereinstimmung gefunden wurde, oder keinmal gewertet werden. Das bedeutet, dass jeder Teil einer Zeichenkette höchstens Bestandteil einer einzigen Übereinstimmung sein kann. Dies hat folgende Auswirkung: Wenn ein Plagiator zweimal einen gleichen Abschnitt kopiert, wird der kopierte Abschnitt nur einmal als plagiiert gewertet. Andernfalls könnte ein Programm, welches einen Abschnitt eines anderen Programms mehrfach beinhaltet, ein hundertprozentiges Plagiat des anderen Programms darstellen, obwohl nur ein Teil kopiert worden ist.

Anforderung 2: Translokationsresistenz
Das Verschieben von Teilen einer Zeichenkette darf den bestimmten Ähnlichkeitswert nicht ändern. Daraus folgt, dass das Verschieben einzelner Methoden in einem Softwareplagiat keine Änderung am berechneten Ähnlichkeitswert hervorrufen darf. Wise

[Wis94, S. 1] ist der Meinung, dass die Auswirkung auf den Ähnlichkeitswert nur *„möglichst klein“* sein muss. Diese Auswirkung gänzlich auszuschließen führt allerdings zu stabileren Ähnlichkeitswerten.

Anforderung 3: Insertions- und Deletionsresistenz
Zusätzliche Einfügungen oder Entfernungen aus einer Zeichenkette, welche nicht plagiierten Stellen betreffen, dürfen keinen Einfluss auf die gefundenen plagiierten Teile einer Zeichenkette haben.

Diese Anforderungen werden, wie sich im Folgenden noch zeigen wird, nur teilweise durch den *GST* erfüllt.

2.2.2 Begriffe und Notation

In diesem Abschnitt werden grundlegende Begriffe geklärt, welche zum Verständnis des *GST* notwendig sind:
Im Folgenden seien P und T zwei zu vergleichende Zeichenketten.

Match
Ein Match beschreibt die zeichenweise Übereinstimmung eines Substrings, welcher am Index p von P mit dem Zeichen P_p beginnt, mit einem Substring, welcher am Index t von T mit dem Zeichen T_t beginnt. Der Match endet entweder vor zwei nicht übereinstimmenden Zeichen, vor dem Ende von T oder P oder vor einem *markierten Zeichen*. Die Markierung wird in der Definition von Tile beschrieben. Matches werden vom *GST* vorübergehend erstellt. Ein Zeichen kann in mehreren Matches vorkommen („eins zu viele“-Beziehung). Ein Match wird wie folgt dargestellt: $match(p, t, s)$. Dabei ist s die Länge der Übereinstimmung.

Maximal Match
Ein Maximal Match ist ein Match mit größtmöglicher Länge.

Tile
Ein Tile wird aus einem Maximal Match erstellt, indem alle Zeichen von T und P, welche der Maximal Match beinhaltet, markiert werden. Markiert bedeutet, dass zwischen den markierten Zeichen und nicht markierten Zeichen keine Übereinstimmung mehr gefunden werden kann. Ein Tile besteht dauerhaft. Jedes Zeichen darf nur in exakt einem Tile vorkommen („eins zu eins“-Beziehung). Ein Tile der Länge s, welcher am Index p von P und am Index t von T beginnt, wird als $tile(p, t, s)$ geschrieben.

Minimum Match Length (*MML*)
Die *MML* gibt die Mindestlänge aller Matches und Tiles an. Sie muss größer 0 sein. Übereinstimmungen im Bereich weniger Zeichen (z.B. mit der Länge 1-3) kommen

sehr häufig vor und sind wenig signifikant. Deshalb wird die *MML* genutzt um kleine Tiles zu verhindern.

2.2.3 Algorithmus

Das Ziel des *GST* ist es, alle identischen Substrings nach den in Abschnitt 2.2.1 erklärten Anforderungen zur Ähnlichkeitsbestimmung zu bestimmen. Diese Substrings müssen sowohl in P) als auch in T vorkommen. Zusätzlich zu den oben genannten Anforderungen geht der *GST* davon aus, dass lange Tiles im Bezug auf plagiierte Stellen aussagekräftiger sind als kurze Tiles. Ein Grund hierfür ist, dass lange Tiles weniger zufällig entstehen. Um letztere Anforderung zu erfüllen, ist der *GST* „gierig“ (egl. greedy) entworfen, also so, dass er die Tiles in absteigender Größe findet.

Algorithmus 1: Greedy String Tiling Algorithmus

```
1  Quellen: [Wis94, S 4], [Mül07, S 81]
   Daten : Zeichenkette T, Zeichenkette P, MML
   Ergebnis : Menge aller gefundenen Tiles
2  tiles = {}
3  wiederhole
4      maxMatchLength = MML
5      currentMaxMatches = {}
6      für jedes unmarkierte Zeichen P_p in P tue
7          für jedes unmarkierte Zeichen T_t in T tue
8              j=0
9              solange P_{p+j} == T_{t+j} ∧ unmarkiert(P_{p+j}) ∧ unmarkiert(T_{t+j}) tue
10                 j=j+1
11                 wenn j == maxMatchLength dann
12                     currentMaxMatches = currentMaxMatches ⋃ {match(p,t,j)}
13                 sonst wenn j>maxMatchLength dann
14                     currentMaxMatches = {match(p,t,j)}
15                     maxMatchLength = j
16     für jedes match(p,t,maxMatchLength) ∈ currentMaxMatches tue
17         wenn keine Überschneidung für match(p,t,maxMatchLength) dann
18             für i=0 bis maxMatchLength -1 tue
19                 markiere(P_{p+i})
20                 markiere(T_{t+i})
21             tiles=tiles ⋃ {match(p,t,maxMatchLength)}
22 bis maxMatchLength == MML
23 return tiles
```

Der im folgenden referenzierte Pseudocode des *GST*, findet sich unter Algorithmus 1. Der *GST* lässt sich sich in zwei Phasen aufteilen:

Phase 1 In dieser Phase von Zeile 6 bis 15 bestimmt der *GST* alle Maximal Matches. *MaxMatchLength* verweist auf die Länge der bisher größten gefundenen Matches. Diese hat mindestens die Größe der *MML*. Durch die Iteration der ersten beiden verschachtelten Schleifen über alle unmarkierten Zeichen in P und T (Zeile 6, 7) und der größtmöglichen Vergrößerung der Übereinstimmung durch die dritte verschachtelte Schleife (Zeile 9), werden alle möglichen Matches erkannt. Falls ein größerer Match gefunden wird als alle vorhergehenden, wird dieser Match das einzige Element der Menge *currentMaxMatches*. Zusätzlich wird die *maxMatchLength* auf dessen Länge gesetzt. Wird ein Match gefunden, dessen Länge der aktuellen *maxMatchLength* entspricht, wird er der Menge *currentMaxMatches* hinzugefügt. Nach der Iteration über alle Zeichen in P und T beinhaltet die Menge *currentMaxMatches* alle Maximal Matches.

Phase 2 Nachdem in der ersten Phase alle Maximal Matches bestimmt wurden, wird in dieser Phase über alle gefundenen Maximal Matches iteriert (Zeile 16-21). Für jeden Maximal Match wird überprüft, ob er sich mit einem bisher erkannten Tile überschneidet. Es liegt eine Überschneidung vor, wenn ein Zeichen von P_p bis $P_{p+maxMatchLength}$ und von T_t bis $T_{t+maxMatchLength}$ markiert ist. Falls keine Überschneidung gefunden wird, werden alle Token, welche der Maximal Match umfasst, markiert. Der so neu entstandene Tile wird zu der Menge *tiles* hinzugefügt. Falls eine Überschneidung vorliegt, wird der aktuelle Maximal Match ignoriert und mit der Überprüfung des nächsten Maximal Match fortgefahren.

Wiederholung der Phasen Die beiden Phasen werden solange wiederholt, bis die *maxMatchLength* nicht mehr verändert wird. Das bedeutet, dass im letzten Durchlauf kein Match, welcher größer als die *MML* ist, gefunden worden ist. Durch die fortlaufende Bestimmung der Maximal Matches in der ersten Phase wird sicher gestellt, dass die Tiles in absteigender Größe gefunden werden.

Auf Grund der eindeutigen Zuordnung der Zeichen zu exakt einem Tile wird die Anforderung „Eindeutigkeit“ aus Abschnitt 2.2.1 erfüllt. Auch die Anforderung „Insertions- und Deletionsresistenz“ wird erfüllt, da der *GST* alle Übereinstimmungen zwischen P und T findet, auch wenn zusätzliche Zeichen eingefügt oder entfernt worden sind.
Die Anforderung der Translokationsresistenz wird nicht immer erfüllt.
Um dies zu zeigen sei:

$$
\begin{aligned}
MML &= 2 \\
P &= aadabc \\
T_1 &= adabc \\
T_2 &= daabc
\end{aligned}
$$

Der *GST* findet zwischen P und T_1 die Übereinstimmung *{adabc}*. In T_2 ist das *„a“* von der erster Stelle in T_1 an die zweite Stelle verschoben worden. Der Vergleich von

P und T_2 liefert die Übereinstimmung *{abc}*. Die summierte Länge der Tiles hat sich durch die Transposition von „*a*“ von 5 auf 3 verkleinert, obwohl sie nach Anforderung 3 hätte 5 bleiben müssen. Dieses Problem besteht nur bei einer *MML* größer 1. Bei einer *MML* gleich 1 werden die Übereinstimmungen *{abc}*, *{a}* und *{d}* mit der Gesamtlänge 5 gefunden. Allerdings sind die Auswirkungen einer Translation bei einer *MML* größer 1 und langen Zeichenketten recht gering, da sich die Gesamtlänge der gefunden Übereinstimmungen nur verändert, wenn durch die Verschiebung Teile der ursprünglichen Übereinstimmung abgespalten werden, welche kleiner als die *MML* sind. Werden Teile abgespalten, die größer als die *MML* sind, werden diese wieder als eigene Tiles erkannt. Auf einen Ansatz zu Lösung dieses Problems auf Methodenebene wird in Abschnitt 3.3.2 eingegangen.

Für eine *MML* von 1 ist der *GST* optimal.
Das kann daran nachvollzogen werden, dass der *GST* für jedes Zeichen in P ein übereinstimmendes Zeichen in T findet, falls eine Übereinstimmung vorhanden ist. Letzteres stellen die Schleifen in den Zeilen 6 bis 9 sicher, da hier jedes Zeichen aus P mit jedem aus T verglichen wird und gefundene Übereinstimmung größtmöglich erweitert werden.
Allerdings verhält sich der *GST* bei *MMLs* größer als eins nicht unbedingt optimal. Das bedeutet, dass der *GST* unter dieser Bedingung nicht immer die maximale Abdeckung von P und T mit Tiles findet. Verantwortlich hierfür ist die Eigenschaft des *GST*, dass die Tiles in absteigender Größe gefunden werden.
Zur Veranschaulichung sei:

$$\begin{aligned} MML &= 2 \\ P &= dadabc \\ T &= abcdadbadab \end{aligned}$$

Der *GST* findet zwischen P und T die Übereinstimmung *{adab}*. Die optimale Abdeckung ist aber durch *{dad}* und *{abc}* gegeben, da die Länge aller gefundenen Tiles dann 6 und nicht 4 ist.

Der *GST* besitzt eine Komplexität von $O(n^3)$. Ein Beweis der Komplexität wird in dieser Arbeit nicht aufgeführt, kann aber in [Wis94, S. 5] gefunden werden.
Da der *GST* im realen Einsatz durch seine hohe Komplexität schlecht zu gebrauchen ist, existieren verbesserte Formen des Algorithmus. Zu nennen sind hier der Tuned Greedy String Tiling Algorithmus und der Running-Karp-Rabin Greedy String Tiling Algorithmus. Letzterer weist eine in der Praxis nahezu lineare Komplexität auf. Beide Algorithmen sind in [Wis94, S 5-9] ausführlich beschrieben.

2.2.4 Ähnlichkeitsmaße

Eine Liste aller gefundenen Tiles ist gut dazu geeignet, die plagiierten Stellen zweier Zeichenketten zu analysieren. Allerdings wäre es wünschenswert, zu einer Liste von Tiles Ähnlichkeitsmaße zu bestimmen, welche Aussagen darüber geben, in wie weit sich zwei Zeichenkette ähneln. Die hier beschriebenen Maßeinheiten sind aus [Mül07, S 82-84] übernommen. Die Größen dieser Maßeinheiten werden **Ähnlichkeitswerte** genannt.

$$TotalTileLength(P,T) = \sum_{match(p,t,length) \in tiles} length$$

Die *Total Tile Length (TTL)* ist die summierte Gesamtlänge aller Tiles. Sie stellt die absolute Menge an übereinstimmenden Zeichen zwischen P und T dar. Anders ausgedrückt ist sie die Größe des plagiierten Anteils von P und T. In Abschnitt 2.3 wird darauf eingegangen, wie die TTL dazu verwendet werden kann, vorgegebene Zeichenketten zu filtern.

$$Sim(P,T) = \frac{2 \cdot TotalTileLength}{|P| + |T|}$$

Als relatives Maß gibt $Sim(P,T)$ den Anteil der doppelten Übereinstimmung zweier Zeichenketten im Bezug auf die Summe der Gesamtlängen der Zeichenketten an. Dieses Maß ist nur dann sinnvoll zu verwenden, wenn beide Zeichenketten eine ähnliche Länge haben. Um dies zu demonstrieren sei

$$\begin{aligned} MML &= 2 \\ P &= abc \\ T &= abcdefghjklmnop \end{aligned}$$

Hier wird der Tile {abc} vom GST gefunden. Des Weiteren ist P ein vollständige Kopie von T. Dennoch beträgt $Sim(P,T)$ nur $\frac{2 \cdot 3}{3+15} = 0.3\overline{3}$.
Um dieses Defizit zu umgehen wird der Wert $Sim(T)$ wie folgt definiert:

$$Sim(T) = \frac{TotalTileLength}{|T|}$$

$Sim(T)$ ist ein relatives Maß, welches das Verhältnis zwischen dem plagiierten Anteil zweier Zeichenketten und der Gesamtlänge einer Zeichenkette angibt. Das gleiche Maß kann auch für P bestimmt werden: $Sim(P)$. Beide Maße auf das vorherige Beispiel angewandt ergeben: $Sim(T) = \frac{3}{15} = 0.2$ und $Sim(P) = \frac{3}{3} = 1$. Das bedeutet, dass P zu 100% von T und T zu 20% von P plagiiert ist. Letzteres zeigt, dass die gemeinsame Betrachtung von $Sim(T)$ und $Sim(P)$ ein besser interpretierbares Ergebnis liefert als $Sim(P,T)$. Der Wertebereich der Maße $Sim(T)$ und $Sim(P,T)$ liegt zwischen 0 und 1. Die Werte werden als prozentuale Ähnlichkeiten interpretiert.

2.3 Erkennung von Vorlagen anhand der Total Tile Length

Studentische Abgaben bauen häufig auf einer zu ergänzenden Vorlage auf. Im Folgenden wird ein Ansatz beschrieben, wie bei einer größeren Menge von Zeichenketten eine Vorlage gefiltert werden kann. Es wird davon ausgegangen, dass Zeichenketten existieren, welche keine Plagiate sind und dass die Vorlage von den Studierenden nur ergänzt, aber nicht verändert worden ist. Zur Erkennung der Plagiate werden über das kartesische Produkt alle möglichen zweier Kombinationen aller Zeichenketten erstellt. Daraufhin wird die TTL von jeder Kombination bestimmt. Als nächstes werden die $TTLs$ der Größe nach absteigend sortiert und beispielsweise wie in Diagramm 2.1 dargestellt. Die sortieren $TTLs$ nähern sich einem Grenzwert an. Um mögliche Plagiate

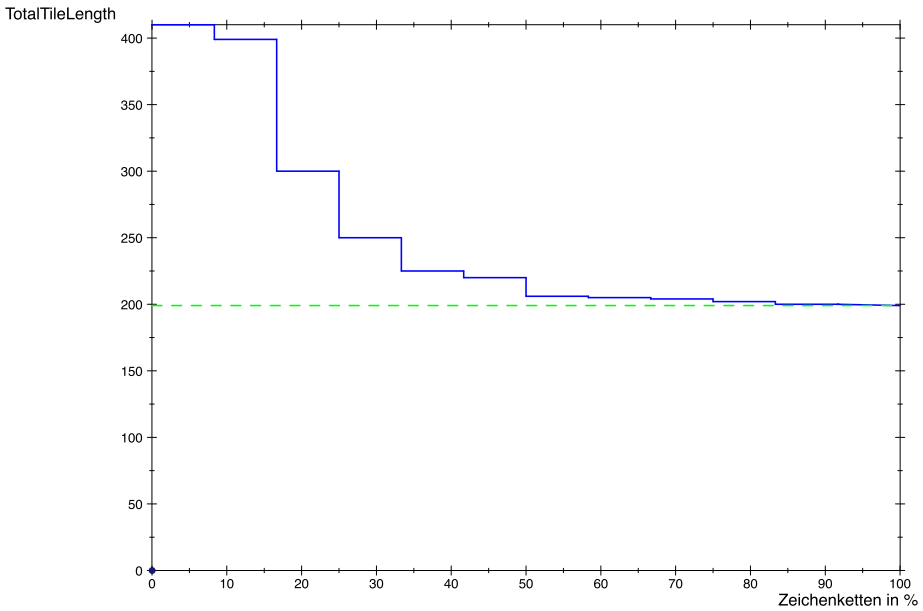


Abbildung 2.1: Beispielplot zum Erkennen von Vorlagen. Blau: Das Verhältnis zwischen der TTL zu Programmen in %. Grün: die Basislinie bei x=199. 17% der Zeichenketten haben eine $TTL > 300$. Diese Zeichenketten sind zu 3/4 plagiiert.

zu finden, betrachtet man alle $Sim(T)$ und $Sim(P)$ der Kombinationen, deren TTL deutlich größer als der bestimmte Grenzwert ist. Da die meisten Zeichenketten nicht plagiiert sind, beschreibt der Grenzwert die Größe der vorgegebenen Zeichenkette. Ein Wert kleiner als die vorgegebene Zeichenkette wird dadurch ausgeschlossen, dass die vorgegebene Zeichenkette nicht geändert werden darf.

2.4 Java-Bytecode

In diesem Abschnitt wird auf den Java-Bytecode sowie auf andere nötige Bestandteile der Klassendateien eingegangen. Dies ist notwendig um den Bytecodevergleich von *PF* nachvollziehen zu können. Eine detaillierte Beschreibung des Bytecodes findet sich in der Spezifikation der *Java Virtual Machine (JVM)* [Lin+11]. Zum besseren Verständnis wird in Abschnitt 2.4.5 ein Beispiel eines Quellcodes und des daraus abgeleiteten Bytecodes aufgeführt.

2.4.1 Klassendateien

Klassendateien (egl. classfiles) mit der Endung „.class“ entstehen durch das Kompilieren von *Java-Quellcodedateien (egl. sourcecodefiles)* mit der Endung „.java“. Klassendateien werden von der *JVM* gelesen und durch einen Übersetzer in systemabhängige Maschinenbefehle überführt [Lin+11, S. 39]. Klassendateien sind eine kompakte, auf das Wesentliche reduzierte, systemunabhängige und schnell ausführbare Darstellung des Java-Quellcodes. Jede Quellcodeklasse wird in eine separate Klassendatei übersetzt. Das gilt auch für interne Klassen. Ein ausführbares Java-Programm liegt in einer „.jar“-Datei vor. Eine solche Datei muss alle Klassendateien eines Programms enthalten. Zusätzlich kann sie die Quellcodedateien beinhalten. Die für diese Arbeit wichtigen Elemente einer Klassendatei sind die *JVM-Instruktionen (egl. JVM instructions)*, die *Konstantentabelle (egl. constant pool)*, die Darstellung der Methoden und die *Zeilennummerntabelle (egl. line number table)*.

2.4.2 Instruktionen der Java Virtual Machine

Die Gesamtmenge aller JVM-Instruktionen eines Programms stellen den Bytecode eines Programms dar. Eine JVM-Instruktion, auch Bytecode-Instruktion genannt, besteht aus einem *Befehlscode (egl. opcode)*, welcher die auszuführende Operation beschreibt. Auf den Befehlscode können *Operanden (egl. operants)* folgen. Diese verweisen auf Daten, welche vom Befehlscode benötigt werden [Lin+11, S. 41]. JVM-Instruktionen beschreiben einfache, maschinencodeähnliche Operationen. Zur besseren Lesbarkeit werden Befehlscodes meist mnemonisch dargestellt. Zum Beispiel hat der Befehlscode „25“ die Mnemonik „aload“ und hat die Funktion eine lokale Variable zu laden [Lin+11, S. 394]. Die Befehlscodes der JVM-Instruktionen beinhalten die semantische Information der Struktur des kompilierten Quellcodes. Sie beschreiben die Sequenz der auszuführenden Operationen durch die Befehlscodes. Zusätzlich verweisen sie durch ihre Operanden auf den Wortlaut des Quellcodes, wie beispielsweise auf den Name einer lokalen Variable oder auf ein String-Literal. Anstellte des Begriffs „JVM-Instruktion“ wird im folgenden auch der Begriff „Befehl“ verwendet.

2.4.3 Konstantentabelle

Die *Konstantentabelle (egl. constant pool)* beinhaltet symbolische Darstellungen von Strukturen, welche von Befehlen und anderen Elementen von Klassendateien referenziert werden. Solche symbolischen Darstellungen sind zum Beispiel Klassennamen, Schnittstellennamen, Feldnamen, Methodennamen oder Namen und Typen von lokalen Variablen [Lin+11, S 82,83]. Die symbolischen Darstellungen werden von der *JVM* während der Verlinkungsphase benötigt, um die binäre Strukturen zu finden, auf welche die symbolischen Darstellungen verweisen [Lin+11, S. 337,349]. Man kann sagen, dass die Konstantentabelle die Bezeichnungen der Elemente des Quellcodes und somit den Wortlaut des Quellcodes beinhaltet.

2.4.4 Methoden

Jede Methode wird innerhalb der Klassendatei durch eine eigene Informationsstruktur dargestellt [Lin+11, S. 97]. Die für diese Arbeit wichtigsten Elemente dieser Struktur sind das *Codeattribut (egl. code attribute)* sowie dessen Attribute: *Zeilennummerntabellenattribut (egl. line number table attribute)* und *Variablentabellenattribut (egl. local variable table attribute)*. Das Codeattribut beinhaltet alle Befehle einer Methode [Lin+11, S. 120]. Das Zeilennummerntabellenattribut beinhaltet Informationen, um von einer Menge von Befehlen auf die Quellcodezeile, von welcher diese abgeleitet worden sind, zurückzuschließen [Lin+11, S. 139]. Das Variablentabellenattribut enthält unter anderem den gültigen Bereich für jede lokale Variable einer Methode und einen Index, welcher auf den Inhalt der Variable verweist [Lin+11, S. 140]. JVM-Instruktionen referenzieren auf die Variablentabelle, welche ein Teil der Struktur des Variablentabellenattributs ist.

2.4.5 Beispiel zum Bytecode

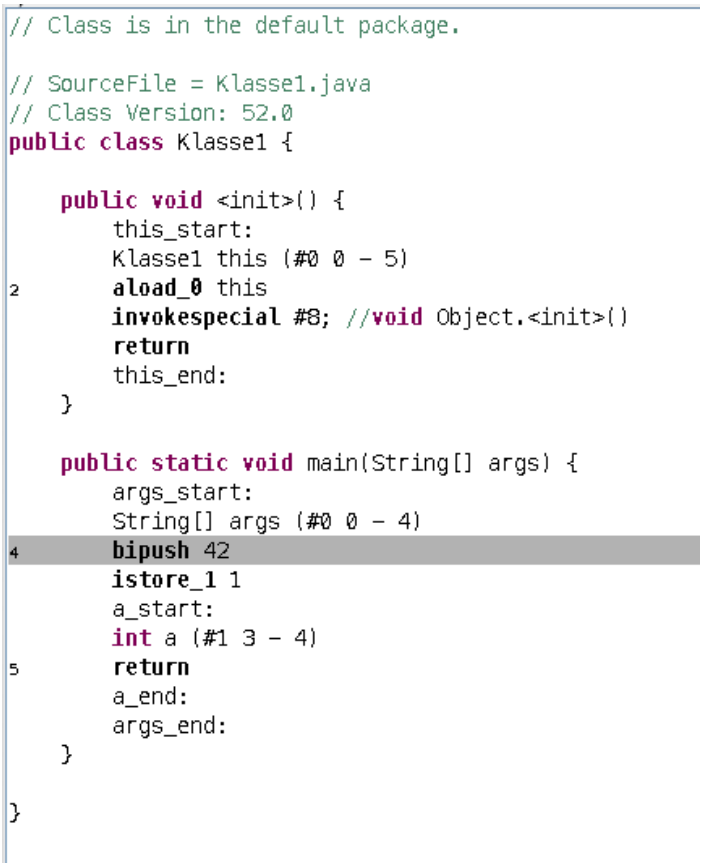
In diesem Beispiel werden die eben vorgestellten Strukturen der Klassendatei veranschaulicht. Die kompilierte Form der Klasse „Klasse 1“ (Abb. 2.2) beinhaltet die JVM-Instruktionen (Abb. 2.4) und die Konstantentabelle (Abb. 2.3). In der Konstantentabelle findet man beispielsweise unter dem Index 2 den Namen „Klasse 1“, unter Index 18 den Namen der lokalen Variable „a“ und unter Index 14 den Namen der Mainmethode. In der Darstellung der JVM-Instruktionen kann gesehen werden, dass die JVM-Instruktion „bipush 42“ den Wert 42 auf den Stack setzt. „istore_1“ lädt den obersten Wert des Stacks auf die lokale Variable an Index 1. Die Befehle „bipush“ und „istore_1“ entstanden durch die Kompilierung der vierten Quellcodezeile.

```
1
2 public class Klasse1 {
3 public static void main(String[] args) {
4   int a = 42;
5 }
6 }
```

Abbildung 2.2: Quellcode der Klasse „Klasse1“

Id	Type	Value
0		[empty]
1	Class reference	Klasse1
2	UTF-8 Text	Klasse1
3	Class reference	java.lang.Object
4	UTF-8 Text	java/lang/Object
5	UTF-8 Text	<init>
6	UTF-8 Text	()V
7	UTF-8 Text	Code
8	Methodref	void java.lang.Object.<init>()
9	Name and type	void <init>()
10	UTF-8 Text	LineNumberTable
11	UTF-8 Text	LocalVariableTable
12	UTF-8 Text	this
13	UTF-8 Text	LKlasse1;
14	UTF-8 Text	main
15	UTF-8 Text	([Ljava/lang/String;)V
16	UTF-8 Text	args
17	UTF-8 Text	[Ljava/lang/String;
18	UTF-8 Text	a
19	UTF-8 Text	I
20	UTF-8 Text	SourceFile
21	UTF-8 Text	Klasse1.java

Abbildung 2.3: Die Konstantentabelle der Klasse „Klasse1“ (Abb. 2.2), dargestellt durch die Software *reJ* [Koi]. „Type“ gibt die Art der Referenz an. „Id“ beinhaltet den Index eines Wertes in der Konstantentabelle. „Value“ beschreibt einen zu einer ID gehörenden Wert.



```
// Class is in the default package.

// SourceFile = Klasse1.java
// Class Version: 52.0
public class Klasse1 {

    public void <init>() {
        this_start:
        Klasse1 this (#0 0 - 5)
2       aload_0 this
        invokespecial #8; //void Object.<init>()
        return
        this_end:
    }

    public static void main(String[] args) {
        args_start:
        String[] args (#0 0 - 4)
4       bipush 42
        istore_1 1
        a_start:
        int a (#1 3 - 4)
5       return
        a_end:
        args_end:
    }

}
```

Abbildung 2.4: Die JVM-Instruktionen der Klasse „Klasse1“ (Abb. 2.2), dargestellt durch die Software *reJ* [Koi]. „#...“ (Bsp. #8), beschreibt eine Referenz in die Konstantentabelle. „(#...)“ verweist auf eine lokale Variable. Die Zahlen am Anfang der Zeilen verweisen auf die ursprüngliche Quellcodezeile. Diese Zahlen sind eine Abbildung aus der Zeilennummerntabelle.

2.5 Byte Code Engineering Library

Die *Byte Code Engineering Library (BCEL)* [Fou] wird in *Plagiarism Finder* zur Handhabung des Bytecodes verwendet. Die *BCEL* bietet die Möglichkeit, Java-Klassendateien zu analysieren, zu manipulieren und zu erzeugen. Die einzelnen Elemente einer Klassendatei werden durch entsprechende Klassen abstrahiert. So gibt es beispielsweise eine „Method-“, eine „Code-“ und eine „LineNumberTable“-Klasse. Eine Erläuterung der Funktion dieser Elemente ist in Abschnitt 2.4.3 aufgeführt.

2.6 Stufen des intelligenten Plagiierens

Dieser Abschnitt beschreibt mehrere Stufen des intelligenten Plagiierens. Diese Stufen sind an die von Faidhi und Robinson definierten Stufen [SKR86, S. 18] angelehnt und in Details ergänzt. Auf jeder Stufe werden Änderungen im Quellcode beschrieben, die ein Plagiator vornehmen kann. Änderungen dieser Art dürfen zu keinen Änderungen in der Menge der bestimmten Tiles führen. Diese Stufen werden hier verwendet, um Softwareplagiatserkennungsprogramme zu klassifizieren. Jede Stufe kann nur dann erreicht werden, wenn jede niedrigere Stufe erfüllt ist.

- Stufe 1: Bearbeiten von Kommentaren und Änderung der Textgestaltung (z.B. Einfügen von Absätzen)
- Stufe 2: Änderung von Kennzeichnungen (z.B. Methodennamen)
- Stufe 3: Änderung der Reihenfolge, Sichtbarkeit oder Veränderbarkeit von Methoden, Datenfeldern, Klassen oder Importe (ohne Sichtbarkeit und Veränderbarkeit)
- Stufe 4: Änderung der Anzahl von Methoden, Datenfelder, Importe oder Schlüsselwörter
- Stufe 5: Änderung von Strukturblöcken (z.B. For- statt While-Schleifen)
- Stufe 6: leichte Änderung der Programmlogik, welche keinen Einfluss auf das Ergebnis hat (z.B. int a = 1; a +=1; statt int a=2;)

In der Menge von Tiles sollte es zu Änderungen kommen, wenn:

- Änderungen im Code zu einem anderen Verhalten des Programms führen.
- das Ergebnis gleich bleibt, aber auf einem anderen Lösungsansatz basiert.

Die Änderungen von Stufe 1 bis 6 werden im Folgenden auch **plagiierte Änderungen** genannt. Änderungen, welche zu einer Änderung in der Menge von Tiles führen, werden **nicht-plagiierte Änderungen** genannt. Wenn im Folgenden von einer Änderung auf einer Stufe die Rede ist, bedeutet dies, dass die Änderung, welche in dieser Stufe neu beschrieben wird, angewendet worden ist, aber nicht, dass alle Änderungen der niedrigeren Stufen angewendet worden sind.

3 Entwurf und Implementierung

In diesem Kapitel wird die entwickelte Plagiatserkennungssoftware *Plagiarism Finder (PF)* vorgestellt. Es wird auf die Anforderungen (Abs. 3.1), die Einordnung (Abs. 3.2), die Funktionsweise (Abs. 3.3), die wesentlichen Klassen (Abs. 3.4), und die Funktionen der Benutzeroberfläche (Abs. 3.5) von *PF* eingegangen.

3.1 Anforderungen

An das entwickelte System wurden folgende Anforderungen gestellt:

- Das Erkennen von Plagiaten muss auf der Basis eines Bytecodevergleichs durchgeführt werden.
- Die Software muss in der Lage sein, „.jar“-Dateien einzulesen und diese auf Plagiate untersuchen können.
- Die Software muss die in Abschnitt 2.2.4 definierten Ähnlichkeitsmaße berechnen und die Ergebnisse exportieren können.
- Die Anforderungen zur Ähnlichkeitsbestimmung aus Abschnitt 2.2.1 müssen erfüllt sein.
- Es muss eine Möglichkeit bestehen, das in Abschnitt 2.3 beschriebene Vorgehen beim Umgang mit Vorlagen durchzuführen.
- Es sollen aktuelle Algorithmen verwendet werden und auf schon vorhandenen Programmierschnittstellen aufgebaut werden.
- Es soll einen Weg geben, die erkannten Übereinstimmungen im Bytecode beziehungsweise im Quellcode einsehen zu können.

3.2 Einordnung

Plagiarism Finder ist ein *SMS* (Abs. 2.1), welches als eine Desktopanwendung umgesetzt worden ist. Die Struktur der zu vergleichenden Programme wird durch eine normalisierte Form des Java-Bytecodes ohne Operanden dargestellt. Natürlichsprachliche Elemente des Codes werden ignoriert. Da die Befehlscodes der JVM-Instruktionen die semantischen Informationen des Quellcodes beinhalten, wird durch einen Vergleich dieser eine Erkennung von intelligenten Plagiaten ermöglicht. Der Algorithmus zum Vergleichen der Repräsentation der Struktur ist der *GST* (Abs. 2.2.3). Die Software ist mit *Java 8* entwickelt worden. Zum Testen wird *JUnit in Version 4.11* verwendet. Zur Extraktion und Bearbeitung des Bytecodes dient die *BCEL* (Abs. 2.5). Die Benutzeroberfläche ist mit *JavaFX 8* erstellt worden.

3.3 Plagiatserkennungsprozess

In diesem Abschnitt wird der Plagiatserkennungsprozess von *PF* beschrieben. Abbildung 3.1 gibt einen Überblick über diesen Prozess.

Zuerst werden alle zu vergleichende Programme eingelesen. Diese Programme müssen in Form von „jar“-Dateien vorliegen. Für jedes Programm wird eine Liste seines Bytecodes erstellt. Hierfür werden alle JVM-Instruktionen jeder Methode in jeder Klasse ausgelesen und in einer Liste aneinandergereiht. Zusätzlich wird bei jeder neuen Klasse und Methode ein Pseudoinstruktionsobjekt in die Liste der Befehle eingefügt. Ein Pseudoinstruktionsobjekt referenziert auf eine Klasse oder Methode. Letzteres ist nötig, damit eine Rückführung von einer JVM-Instruktion zu der sie beinhaltenden Methode und Klasse möglich ist. Der Vergleich eines Pseudoinstruktionsobjekt mit einem beliebigen anderen Objekt liefert immer ungleich. Folglich kann es keine Methoden- und Klassenübergreifende Tiles geben. Näheres hierzu wird in Abschnitt 3.3.2 erläutert.
Als nächstes werden alle Referenzen der JVM-Instruktionen entfernt. Das bedeutet, das jeder Operand, welcher auf ein anderes Element in der Konstantentabelle oder in der Variablentabelle verweist, entfernt wird. Näheres hierzu findet sich in Abschnitt 3.3.1. Einfachheitshalber wird das Entfernen der Referenzen so umgesetzt, dass nur die Befehlscodes der JVM-Instruktionen verglichen werden.
Als nächstes wird über das kartesische Produkt die Menge aller möglichen Programmpaare bestimmt. Für jedes Programmpaar werden die beiden Befehlslisten mit Hilfe des *GST* und einer vom Benutzer ausgewählten *MML* miteinander verglichen. Die *MML* wird in JVM-Instruktionen angegeben. Das Ergebnis des Vergleichs ist eine Menge von Tiles je Programmpaar.
An dieser Stelle teilt sich der Prozess in zwei Teilprozesse auf.

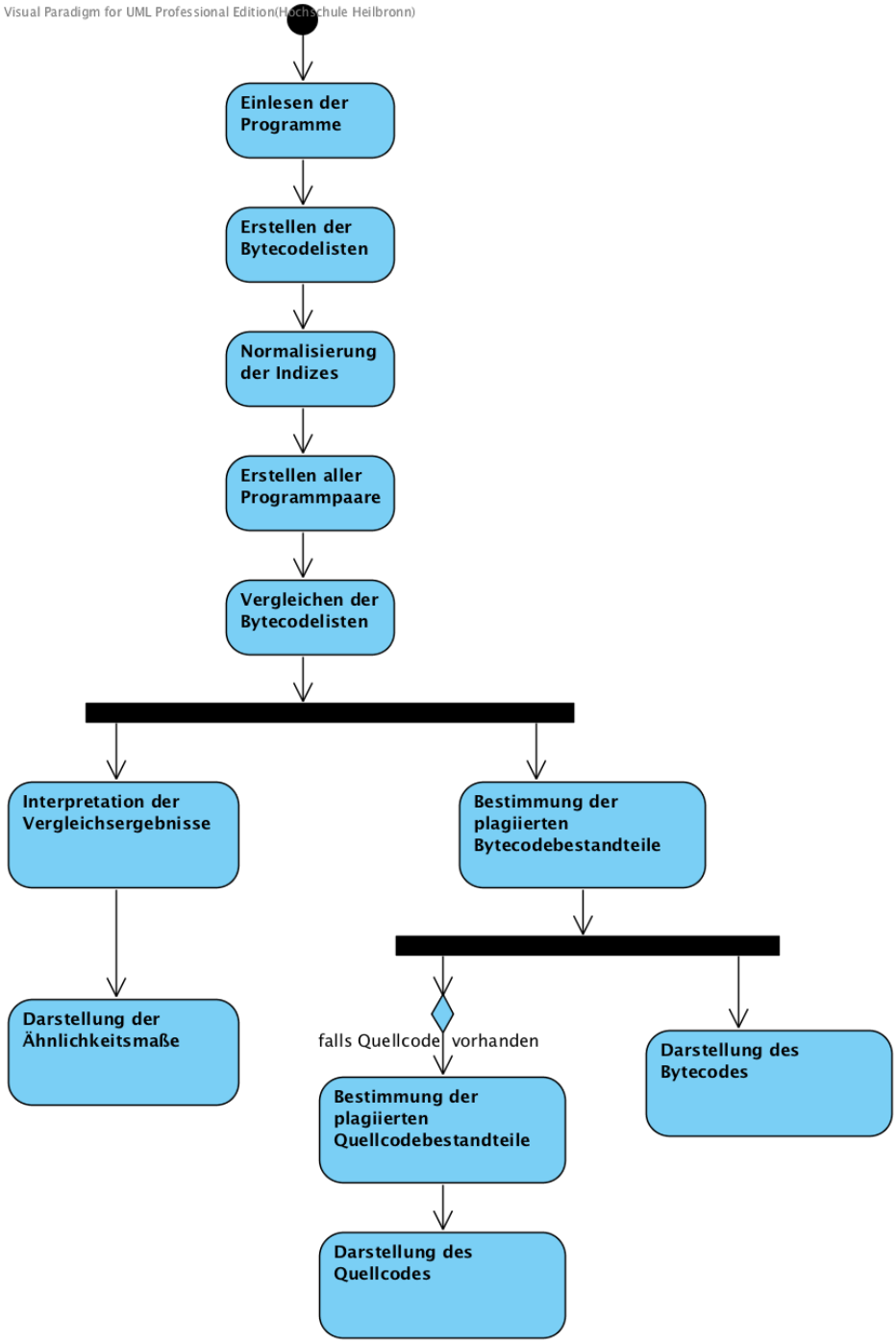


Abbildung 3.1: Der Plagiatserkennungsprozess von *Plagiarism Finder*

Der erste Teilprozess wertet die Menge an Tiles aus. Hierfür werden die Ähnlichkeitsmaße $Sim(p1)$, $Sim(p2)$, $Sim(p1, p2)$ sowie die TTL bestimmt. Bei der Berechnung der Gesamtlänge einer Liste von JVM-Instruktionen werden alle Befehle ignoriert, welche in Methoden vorkommen, deren Menge an Befehlen kleiner als die MML ist.

Hierdurch wird das durch die Verschiebung von Methoden kleinst mögliche Ähnlichkeitsmaß bestimmt. Die Begründung dieses Vorgehens findet sich in Abschnitt 3.3.2. Die ermittelten Werte werden sowohl in tabellarischer Form als auch in Diagrammen angezeigt.
Der zweite Teilprozess ermittelt zu jedem Tile die von ihm beschriebenen Abschnitte im Bytecode. Zu diesen Abschnitten werden die sie beinhaltenden Methoden und Klassen bestimmt. Diese Bestimmung wird über die eingefügten Pseudoinstruktionen ermöglicht. Aus den plagiierten Abschnitten des Bytecodes werden die entsprechenden Quellcodeabschnitte über die Zeilennummerntabelle abgeleitet. Zuletzt wird für den Bytecode sowie den Quellcode jeweils eine Ansicht erzeugt, welche den kompletten Code jedes Programms mit hervorgehobenen plagiierten Anteilen darstellt.

3.3.1 Näheres zu Bytecodenormalisierung

Wie im Prozess beschrieben werden alle Referenzen der JVM-Instruktionen entfernt. Dies ist notwendig, da kleine Änderungen im Quellcode zu abgeänderten Indizes in der Konstantentabelle und in der Variablentabelle führen [BU98, S. 6]. Demnach unterscheiden sich bei Änderungen die Referenzen der JVM-Instruktionen auf diese Indizes. Die Referenzen auf die Konstantentabelle können entfernt werden, da diese die Informationen über die Namen von Strukturen enthält und in dieser Software nur die Semantik der Strukturen verglichen werden soll. Die Entfernung der Referenzen in die Variablentabelle hat eine Auswirkung auf die Funktionsweise des Programms. Die Ursache ist, dass Befehle Operationen auf einer bestimmten lokalen Variable durchführen. Es ist meist nicht möglich zu bestimmen, welche Variable einer anderen Variablen in einem anderen Programm entspricht, da diese sowohl einen anderen Typ als auch einen anderen Namen haben kann. Deshalb kann nicht verhindert werden, dass hier Information über die Funktionsweise verloren gehen.
Beka und Udi stellen in [BU98, S. 5,6] eine weitere Möglichkeit vor, wie mit den Referenzen der JVM-Instuktionen umgegangen werden kann. Im Folgenden wird die Funktionsweise dieser Methode dargestellt. Zusätzlich wird aufgezeigt, warum sie in diesem Kontext nicht angewendet werden kann.
Diese Methode ersetzt jede Referenz mit dem Abstand zur letzten identischen Referenz. Wenn eine Referenz das erste Mal vorkommt, wird sie durch eine 0 ersetzt. Das folgende Beispiel veranschaulicht die Modifikationen. Seien die Buchstaben Befehlscodes und die Zahlen dahinter Referenzen:

Nach Beka und Udi:
$a12b15ca15b12$ geht über in $a0b0ca3b7$
$a12b15cca15b12$ geht über in $a0b0cca4b8$

Nach dieser Arbeit:
$a12b15ca15b12$ geht über in $abcab$
$a12b15cca15b12$ geht über in $abccab$

Bei Vergleich von „$a0b0ca3b7$“ und „$a0b0cca4b8$“ mit dem *GST* wird die Übereinstimmung von „$a15b12$“ nicht erkannt. Vergleicht man „$abcab$“ mit „$abccab$“, wird unter anderem die Übereinstimmung „ab“, welche von „$a15b12$“ abgeleitet worden ist, gefunden. Aus dem Beispiel lässt sich erkennen, dass die Methode von Beka und Udi in Kombination mit dem *GST* nicht sinnvoll eingesetzt werden kann.

3.3.2 Vorgehen zur Einhaltung der Translokationsresistenz

Wie in Abschnitt 2.2.3 beschrieben werden die Anforderungen „Eindeutigkeit“ und „Insertions- und Deletionsresistenz“ (Abs. 2.2.1) allein durch den *GST* erfüllt. Die Anforderung „Translokationsresistenz“ wird nicht erfüllt. Diese Anforderung lautet: „Das Verschieben von Teilen einer Zeichenkette darf das bestimmte Ähnlichkeitsmaß nicht ändern“. Folgend wird der Lösungsansatz beschrieben, mit welchem diese Anforderung durch *PF* auf Methodenebene umgesetzt wird. Ein solcher Teil einer Zeichenkette wird in diesem Kontext durch eine Sequenz von JVM-Instruktionen dargestellt. Allerdings verändert das Verschieben einzelner JVM-Instruktionen meistens das Ergebnis des ausgeführten Programms. Diesbezüglich ist es erwünscht, dass sich durch Verschieben der JVM-Instruktionen der Ähnlichkeitswert ändert. Das Verschieben der Befehlssequenz einer Methode ist dagegen möglich, ohne dass sich das Ergebnis des Programms ändert. Allerdings kann das Verschieben von Methoden den Ähnlichkeitswert ändern. Dies geschieht, falls eine Methode, deren Befehlssequenz kürzer ist als die *MML*, an die Befehlssequenz einer Methode anschließt, welche länger als die *MML* ist und beide Befehlssequenzen Teil eines plagiierten Abschnittes sind. Letzteres wird in Abschnitt 3.2 veranschaulicht.
Dieses Problem kann gelöst werden, indem keine methoden- oder klassenübergreifende Tiles erlaubt werden. Daraus folgt, dass das Verschieben von Methoden keinen Einfluss auf die Tiles haben kann. Der so berechnete Ähnlichkeitswert stellt das Minimum aller Ähnlichkeitswerte dar, welche durch Verschieben von Methoden erreicht werden können. Dies lässt sich daraus herleiten, dass eine Methode, welche kleiner als die *MML* ist, bis auf wenige Ausnahmen so verschoben werden kann, dass sie nicht Bestandteil eines Tiles ist. Keine methoden- oder klassenüberschneidende Tiles werden

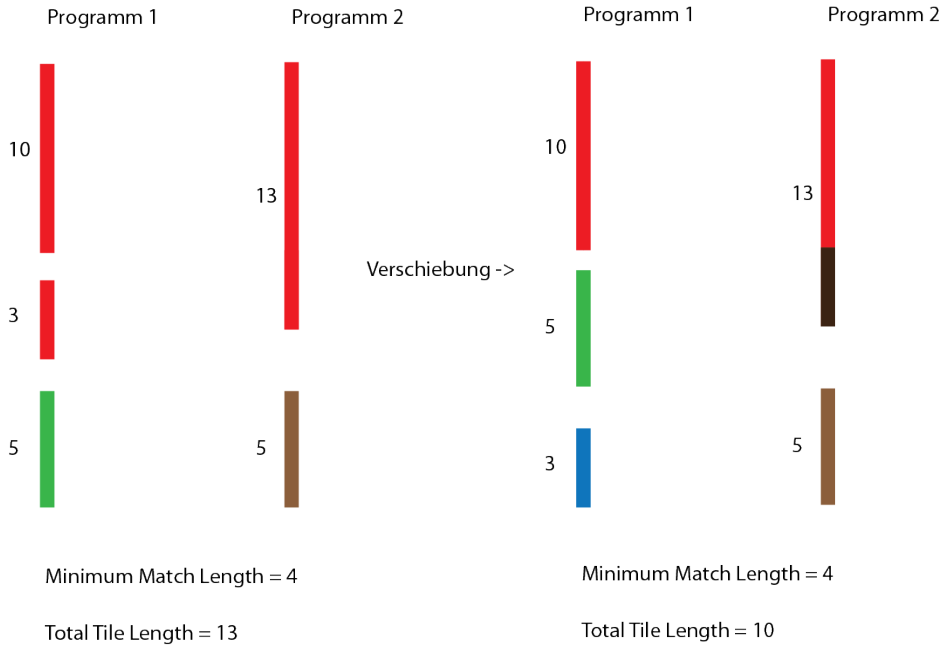


Abbildung 3.2: Änderung der *TTL* durch das Verschieben von Methoden. Die Balken stellen Methoden dar. Die Zahlen vor den Balken beschreiben die Länge der Befehlssequenzen. Die rot eingefärbten Methodenteile stellen als plagiiert erkannte Befehlssequenzen dar. Die Methoden der Länge 5 (grün) und 3 sind verschoben worden.

erreicht, indem die in die Liste mit Befehlen eingefügten Pseudoinstruktionsobjekte bei Vergleich mit beliebigen anderen Objekten immer ungleich liefern.
Mit dieser Lösung werden alle Methoden, deren Befehlssequenzen kleiner als die *MML* sind, vom *GST* ignoriert. Es ist bei Existenz einer solchen Methode nicht möglich, einen Sim-Wert von 1 zu erhalten, da höchstens das ganze Programm außer einer solchen Methode als plagiiert erkannt werden kann. Damit dies möglich ist, wird in *PF* die Gesamtlänge eines Programms über die Menge der Befehle der Methoden bestimmt, deren Befehlssequenz länger als die *MML* ist.

3.4 Entwurf

PF ist als Desktopanwendung entwickelt worden. Die Implementierung orientiert sich an Joshua Blochs Regeln zur Entwicklung von korrekter, robuster und wiederverwendbarer Java-Software [Blo08].

3.4.1 Parallelisierung der Vergleiche

Plagiarism Finder ist mit Hilfe der Programmierschnittstelle *„Stream“* aus Java 8 parallelisiert worden. Jeder einzelne Vergleich zweier Java-Programme kann bei genug Prozessorkernen in einen eigenen Thread ausgelagert werden. Die Menge der Threads wird durch *Stream* selbständig und in Abhängigkeit vom zugrundeliegenden Hardwaresystem bestimmt.

3.4.2 Die wesentlichen Klassen

Im Folgenden werden die wichtigsten Klassen von *PF* vorgestellt. Ein Klassendiagramm mit den Abhängigkeiten der Modellklassen findet sich im Anhang A.1.
Die zentralen Modellklassen sind:

bcel.generic.Instruction:	die Abstraktion von einer JVM-Instruktion durch die *BCEL*.
InstructionList:	eine Liste aus *bcel.generic.Instructions* Objekten. Sie stellt die zu vergleichende Befehlsliste aller Befehle eines Programms dar.
TokenChainSection:	beschreibt einen plagiierten Bereich auf einer *InstructionList*.
Tile:	stellt eine Tile, wie in Abschnitt 2.2.2 definiert, dar. Beinhaltet zwei *TokenChainSections*.
Program:	beinhaltet alle Klassen- und Quellcodedateien einer „.jar“ Datei.
CompareResult:	beschreibt ein Programmpaar und die Menge aller Tiles, welche durch Vergleich der beiden Programme ermittelt worden sind.
ResultInterpretation:	dient zur Darstellung der Werte der drei Sim-Ähnlichkeitsmaße und der *TTL*.

Die zentralen Verarbeitungsklassen sind:

GreedyStringTiling:	generische Implementierung des *GST*. Wird zum Vergleichen zweier *InstructionLists* verwendet.
ProgramLoader:	regelt das Laden von Programmen. Hierzu parst die Klasse aus einer „.jar“-Datei ein *Program*-Objekt.

InstructionListUtil:	erstellt aus einem *Programm* eine *InstructionList*. Fügt hierbei zusätzliche Methoden- und Klassenpseudoinstruktionen ein.
Comparator:	Vergleicht eine Liste von Programmen. Bildet dazu alle Programmpaare und die *InstructionLists* zu jedem Programm. Bestimmt je Programmpaar ein *CompareResult*-Objekt. Erzeugt einen neuen *bcel.generic.Instruction.Comparator* so, dass die Equals-Methode von *bcel.generic.Instruction* nur die Befehlscodes zweier *bcel.generig.Instruction*-Objekte vergleicht.
Interpreter:	bestimmt die Ähnlichkeitswerte aus einer Liste von Tiles. Berechnet hierzu aus einem *CompareResult*-Objekt ein *ResultInterpretation*-Objekt. Verwendet einen *RemoveMethodsManipulator* um alle Methoden, welche kleiner als die *MML* sind, in der Berechnung der Gesamtlänge eines Programms nicht zu berücksichtigen.
RemoveMethodsManipulator:	entfernt aus der *InstructionList* alle Methoden, welche kleiner als die *MML* sind.
TileTranslator:	stellt die Grundlage für die Darstellung der plagierten Bytecode- und Quellcodeabschnitte dar. Er übersetzt ein *Tile*-Objekt in verschiedene Darstellungen. Dazu werden die *TokenChainSection-Objekte* in einem Tile durch andere Objekte von Subklassen von TokenChainSection ausgetauscht. Diese beinhalten zusätzliche Angaben. Zum Beispiel, an welcher Stelle sich der plagiierter Bereich im Quellcode befindet.

3.5 Funktionen der Benutzeroberfläche

In diesem Kapitel wird auf die Benutzeroberfläche von *PF* und deren Funktionen eingegangen. Wie in Abbildung 3.3 zu sehen, können von *PF* ein oder mehrere Java-Programme geladen werden. Zusätzlich kann die *MML* gesetzt werden. Durch Klick auf *„Find plagiarism“* werden für jedes Programmpaar die Werte der Ähnlichkeitsmaße *Sim(p1)*, *Sim(p2)*, *Sim(p1,p2)* und *TTL* berechnet und wie in Abbildung 3.4 angezeigt. Hierbei steht *„p1“* für das erste Programm und *„p2“* für das zweite Programm eines Programmpaars. Die Ansicht der Ähnlichkeitswerte in einem Diagramm

erhält man durch Klick auf *„Charts“* in der tabellarischen Ansicht der Ähnlichkeitswerte. Das Diagramm $Sim(p1 \cup p2)$ beinhaltet sowohl den Wert von *Sim(p1)* als auch den *Sim(p2)*-Wert jedes Programmpaars. Um eine Vorlage, wie in Kapitel 2.3 beschrieben, zu filtern, können das Diagramm der *TTL* und die Sortiermöglichkeit der Ähnlichkeitswerte in der tabellarischen Ansicht verwendet werden. Des Weiteren kann durch Klick auf *„Export CSV“* eine „.csv“-Datei mit allen berechneten Werten exportiert werden. Ein Klick auf eine Tabellenzeile führt zu einer detaillierten Ansicht des Vergleichs zwischen den Programmpaaren (Abb. 3.6). Plagierte Abschnitte sind gleich eingefärbt. Dasselbe gilt für die über die Tableiste auswählbare Quellcodeansicht (Abb. D.1). Durch Klick auf einen Tile wird in beiden Programmen zu der plagiierten Stelle gesprungen. Die Tiles sind in absteigender Größe sortiert. In allen Abbildungn außer in Abbildung 3.5 ist die gleiche Datenbasis verwendet worden.

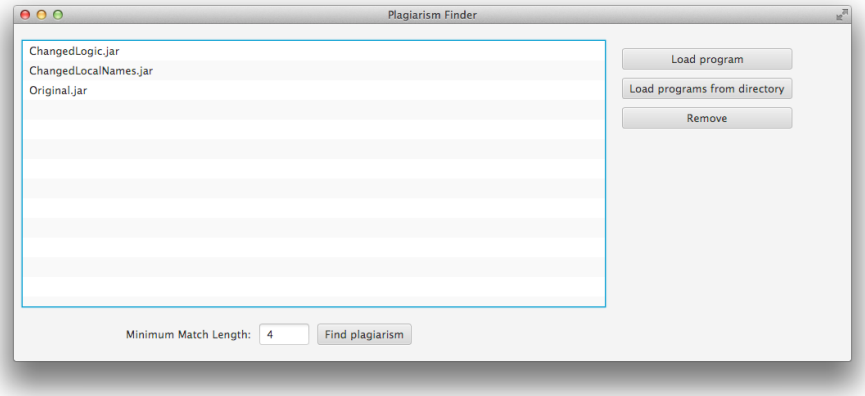


Abbildung 3.3: Programmauswahlansicht von *Plagiarism Finder*.

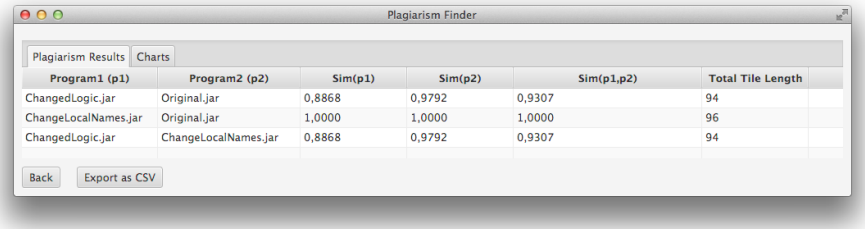


Abbildung 3.4: Ansicht der Ähnlichkeitsmaße und deren Werte in *Plagiarism Finder*.

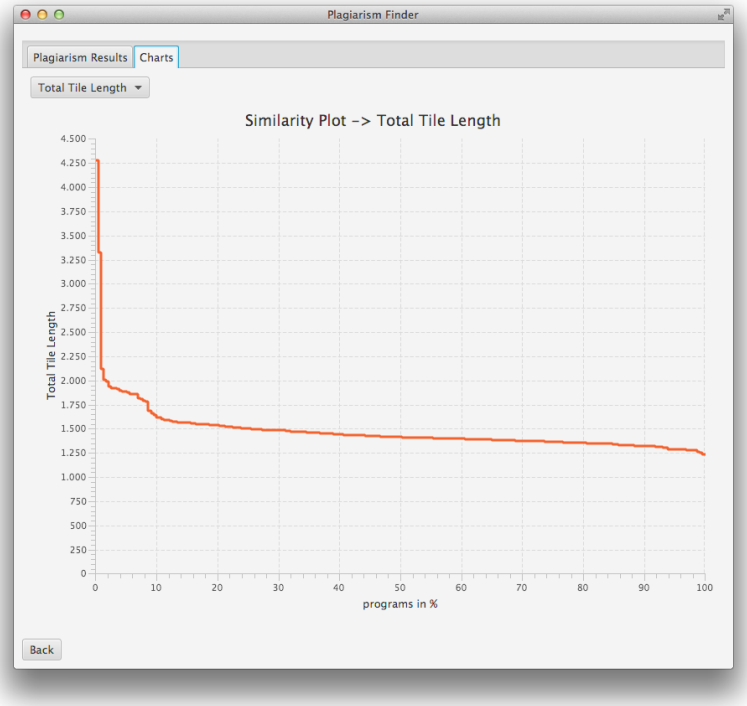


Abbildung 3.5: Ansicht des *TTL*-Diagramms in *Plagiarism Finder*.

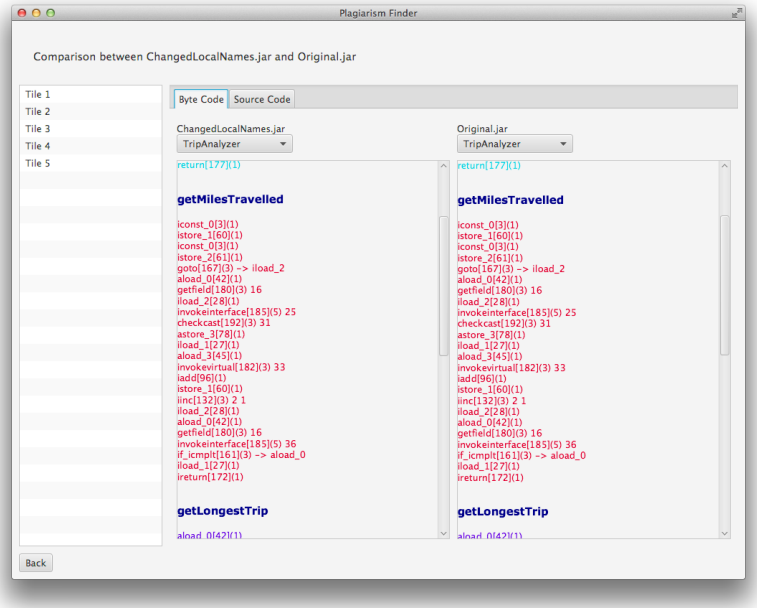


Abbildung 3.6: Bytecodevergleichsansicht in *Plagiarism Finder*. Plagiierte Stellen sind identisch eingefärbt. Jeder Befehl hat die Form: „mnemonic[opcode](Größe in Bytes)“.

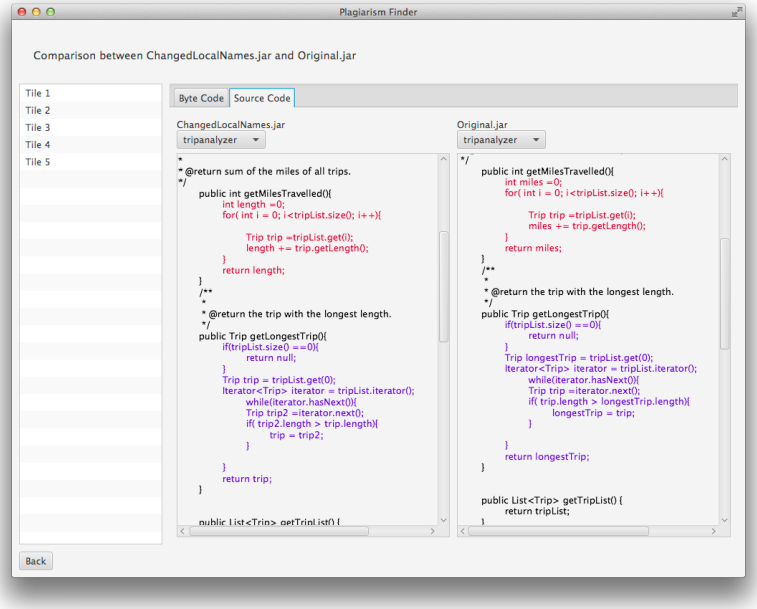


Abbildung 3.7: Quellcodevergleichsansicht in *Plagiarism Finder*. Plagiierte Stellen sind identisch eingefärbt.

4 Evaluation

Dieses Kapitel dient zur Evaluation von *Plagiarism Finder*.
In den ersten beiden Experimenten wird die Stabilität der ermittelten Ähnlichkeitswerte überprüft. Zusätzlich wird analysiert, in welche Stufe des Plagiierens (Abs. 2.6) *PF* einzuordnen ist. Das dritte Experiment dient zur Überprüfung der Praxistauglichkeit von *PF*. Hierzu werden studentische Abgaben einer Vorlesung untersucht. Zuletzt werden im vierten Experiment die Ergebnisse von *PF* mit denen von *JPlag*, eines etablierten Programms zur Softwareplagiatserkennung, gegenübergestellt.
„Stabilität der Ähnlichkeitswerte“ bedeutet in diesem Zusammenhang, dass die *TTL* bei plagiierten Änderungen konstant bleibt und sich die Änderungen der Werte von *Sim(p)* und *Sim(p1,p2)* auf Änderungen der Gesamtlänge der zu vergleichenden Programme zurückführen lassen.
Eine Interpretation und Diskussion der Ergebnisse folgt in Kapitel 5.

4.1 Experiment 1: Evaluierung anhand eigener Beispielprogramme

Das Experiment dient der Analyse, in welche Stufe des Plagiierens (Abs. 2.6) *PF* einzuordnen ist. Hierfür wird die Stabilität der Ähnlichkeitswerte anhand eigens erstellter Beispielprogramme überprüft.

4.1.1 Vorgehensweise

Es werden Plagiate eines Originalprogramms erstellt. Dabei werden plagiierte Änderungen der in Abschnitt 4 vorgestellten Stufen umgesetzt. Da es sich um plagiierte Änderungen handelt, sollten die Ähnlichkeitswerte stabil bleiben. Wenn sich *PF* bei Änderungen nicht stabil verhält, bedeutet dies, dass er Änderungen auf dieser Stufe nicht erkennen kann. Um das in Abschnitt 3.3.2 beschriebene Vorgehen zur Einhaltung der Translokationsresistenz überprüfen zu können, müssen Methoden existieren, welche kleiner als die *MML* sind. Deshalb wird eine *MML* von 4 gewählt. Die erstellten Plagiate können im digitalen Begleitmaterial (Anh. D) gefunden werden.

4.1.2 Ergebnis

Alle ermittelten Ähnlichkeitswerte sind im Anhang C.2 aufgeführt. Ein wesentlicher Ausschnitt der Werte stellt die Tabelle 4.2 dar. Es ist zu erkennen, dass Änderungen auf den Stufen 1, 2 und 3 keinen Einfluss auf die Ähnlichkeitswerte haben. Änderungen der Stufen 4, 5 und 6 haben einen Einfluss auf die Ähnlichkeitswerte. Von Stufe 1 bis 4 verhalten sich die Ähnlichkeitswerte stabil.

Programm 1	Programm 2	sim(p1)	sim(p2)	sim(p1,p2)	TTL	Stufe
ChangedIdentations.jar	Original.jar	1	1	1	96	1
ChangedClassNames.jar	Original.jar	1	1	1	96	2
ChangedMethodOrder.jar	Original.jar	1	1	1	96	3
ChangedNewMethods.jar	Original.jar	0,8496	1	0,9187	96	4
ChangedWhiletoFor.jar	Original.jar	0,787	0,8854	0,8333	85	5
ChangedLogic.jar	Original.jar	0,8868	0,9792	0,9307	94	6

Tabelle 4.2: Repräsentative Auswahl der Ergebnisse des Vergleichs eigener Beispieldaten mit *PF*. Die Spalte *„Stufen“* bedeutet, dass eine Änderung auf dieser Stufe vorgenommen worden ist. Sie bedeutet nicht, dass alle Änderungen niedrigerer Stufen vorhanden sind. Die Namen der Programme in *„Programm 1“* weisen auf die durchgeführten Änderungen hin.

4.2 Experiment 2: Evaluierung durch von Dritten erstellten Plagiate

Dieses Experiment hat das selbe Ziel wie Experiment 1 (Abs. 4.1): zu Überprüfen, bis zu welcher Stufe des Plagiierens (Abs. 2.6) *PF* mit plagiierten Änderungen umgehen kann. Hierfür wird die Stabilität der Ähnlichkeitswerte anhand von Plagiaten überprüft, welche von unabhängigen Plagiatoren erstellt worden sind. Des Weiteren wird überprüft, ob sich die Ähnlichkeitswerte bei nicht-plagiierten Änderungen wie erwartet verändern. Da die in Experiment 1 erstellten Plagiate vom Entwickler der Plagiatserkennungssoftware stammen, kann es zum Auftreten von Bias kommen. Dieses Auftreten soll in diesem Experiment vermieden werden.

4.2.1 Vorgehensweise

Für diesen Versuch wird ein Originalprogramm durch unabhängige Plagiatoren verändert. Zum Erstellen der Plagiate erhalten die Plagiatoren eine ausführliche Anleitung (s. Anh. B.1). Die von ihnen vorgenommenen Änderungen werden in einer externen Datei, also nicht im Quellcode, dokumentiert. Es werden sowohl plagiierte Änderungen als auch nicht-plagiierte Änderungen vorgenommen. Bei ersteren sollten die Ähnlichkeitswerte stabil bleiben, bei zweiteren sollten sich die Ähnlichkeitswerte ändern. Die erstellten Plagiate werden in „.java“-Dateien geliefert. Daraufhin werden diese kompiliert und zusammen mit den erstellten „.class“-Dateien in „.jar“-Dateien gepackt. Die erstellten „.jar“-Dateien werden mit Hilfe von *PF* und einer *MML* von 4 verglichen. Erst nach dem Vergleich wird die Datei mit den dokumentierten Änderungen zum ersten Mal eingesehen. Es wird analysiert, ob sich die von *PF* bestimmten Ähnlichkeitswerte aus den dokumentierten Änderungen herleiten lassen.
Das beschriebene Vorgehen wird gewählt, damit das Experiment schwer zu manipulieren ist und das Auftreten eines Bias unwahrscheinlich ist. Durch dieses Vorgehen wissen die Plagiatoren nichts über die Funktionsweise von *PF*. Zusätzlich sind der Person, welche die Programme durch *PF* vergleichen lässt, die vorgenommenen Änderungen unbekannt. Damit kein Selektionsbias entsteht, sind die dokumentierten Änderungen erst dann einsehbar, nachdem die Ergebniswerte des Vergleichs feststehen.
Die erstellten Plagiate sowie die Dateien mit den dokumentierten Änderungen können im digitalen Begleitmaterial (Anh. D) gefunden werden.

4.2.2 Ergebnis

Alle ermittelten Ähnlichkeitswerte des Experiments sind unter Anhang C.3 einsehbar. Tabelle 4.3 beinhaltet eine repräsentative Auswahl der Ergebnisse. Man kann erkennen, dass Änderungen auf den Stufen 1 und 2 keinen Einfluss auf die Ähnlichkeitswerte haben. Änderungen auf den Stufen 3, 4, 5 und 6 haben meistens einen Einfluss auf die Ähnlichkeitswerte, allerdings gibt es bei Änderungen der Stufen 3, 4 und 6 Ausnahmen. Die Ähnlichkeitswerte verhalten sich bei Änderungen auf den Stufen 1 und 2 sowie bei allen plagiierten Änderungen stabil.

Programm1 (p1)	Sim(p1)	Sim(p2)	Sim(p1,p2)	TTL	Änderung	Stufe
S20.jar	1	1	1	181	Autorennamen im Javadoc geändert	1
G4.jar	1	1	1	181	Umbenennung von Klassen, Methoden, Datenfeldern und lokalen Variablen	2
S3.jar	1	1	1	181	Änderung der Reihenfolge von Methoden, Datenfelder und Konstruktoren	3
S5.jar	0.9779	0.9779	0.9779	177	Alle Zugriffsmodifikatoren auf „package-private“ geändert	3
S6.jar	1	1	1	181	Zusätzliche Importe	4
S8.jar	0.9031	0.9779	0.9390	177	Code in Submethoden Ausgelagert	4
S10.jar	0.9779	0.9779	0.9779	177	While statt For-Schleife	5
S18.jar	1	1	1	181	„x+=y“ durch „x = x+y“ ersetzt	6
R3.jar	0.9676	0.9890	0.9781	179	„Math.pow“ durch „x*x“ ersetzt	6
S9.jar	0.9945	0.9945	0.9945	180	„+=“ in „-=“ geändert	nicht plagiiert
G2.jar	0.2250	0.1492	0.1794	27	Komplette Neuimplementierung der Logik	nicht plagiiert

Tabelle 4.3: Eine repräsentative Auswahl der Ergebnisse des Vergleichs von durch Dritten erstellten Programme mit *PF*. Die Spalte *„Stufen“* bedeutet, dass eine Änderung auf dieser Stufe vorgenommen worden ist. Sie bedeutet nicht, dass alle Änderungen niedrigerer Stufen vorhanden sind. *„Programm 2 (p2)“* ist immer das Programm *„Original.jar“*. Die Stufe *nicht plagiiert* bedeutet, dass eine nichtplagiierte Änderung vorgenommen worden ist.

4.3 Experiment 3: Evaluierung anhand eines Originaldatensatzes

Dieses Experiment dient zur Überprüfung, ob *PF* dazu geeignet ist, Plagiate in studentischen Originalabgaben zu finden.

4.3.1 Vorgehensweise

Zur Durchführung dieses Experiments werden Originalabgaben der Begleitaufgaben zur Vorlesung Grundlagen der Praktischen Informatik im Studiengang Medizinische Informatik der Hochschule Heilbronn und der Universität Heidelberg analysiert. Da die Studenten ihre Aufgabe mehrmals abgeben können, wird immer die letzte eingereichte Abgabe zum Vergleich ausgewählt. Zuerst wird von jedem Studierenden die gemäß § 4 Abs. 1 BDSG notwendige Erlaubnis eingeholt, um ihre Abgaben für den Zweck dieses Experiments verwenden zu dürfen. Die Abgaben liegen in „.jar“-Dateien vor, welche keine kompilierten „.class“-Dateien, sondern nur „.java“-Dateien beinhalten. Die Quellcodedateien werden durch das eigens für dieses Experiment geschriebene Programm *BatchJarCompilerAndAnonymizer* anonymisiert und kompiliert. Das Programm ist von Dipl.-Inform. Med. Hauke Hund und dem Ersteller dieser Arbeit gemeinsam entwickelt worden. *BatchJarCompilerAndAnonymizer* entpackt zuerst alle „.jar“-Dateien. Anschließend werden die entpackten Quellcodedateien anonymisiert. Hierzu werden alle Zeilen, welche die *„@author“* Annotation beinhalten, entfernt. Im Anschluss werden die „.java“-Dateien kompiliert. Zuletzt wird eine „.jar“-Datei erstellt, welche alle „.java“- und „.class“-Dateien beinhaltet. Als nächstes werden alle Abgaben durch *PF* mit einer *MML* von 10 untereinander verglichen, die Ähnlichkeitswerte ermittelt und nach Plagiaten gesucht. Zusätzlich wird analysiert, ob sich der Ansatz zur Filterung von Vorlagen (Abs.2.3) umsetzen lässt.
Die zu vergleichenden Daten, das Programm *BatchJarCompilerAndAnonymizer* und das Formular zur Erlaubniseinholung befinden sich im digitalen Begleitmaterial (Anh. D).

4.3.2 Ergebnis

Alle ermittelten Ähnlichkeitswerte sind im Anhang C.4 aufgelistet.
Der Vergleich wurde auf einem MacBook Pro von Mitte 2010 mit 8 GB 1067 MHz DDR3 RAM und einem 2,4 GHz Intel Core 2 Duo Prozessor durchgeführt. Der Vergleichsprozess dauerte 467,12 Sekunden (7,79 Minuten). Es sind 16 Programme in 120 Kombinationen überprüft worden. Ein durchschnittlicher Vergleich eines Programmpaars dauerte 3,89 Sekunden. Im Schnitt beträgt die von *PF* bestimmte Länge der Programme 3990,69 JVM-Instruktionen.

Die Verteilungen der Werte der Maße TTL und $Sim(p1, p2)$ werden in den Diagrammen 4.1 und 4.2 dargestellt. Die TTL liegt in einem Bereich zwischen 2482 und 2905 Befehlen. 100% der Programme stimmen mit 2482 Befehlen, 50% mit 2650 Befehlen überein. $Sim(p1, p2)$ liegt in einem Bereich zwischen 57% und 76%. 50% der Programme stimmen zu 67% überein.

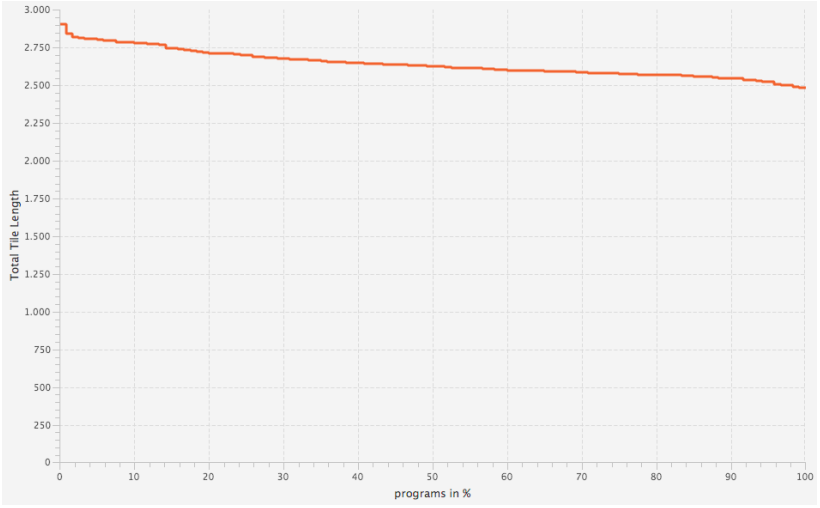


Abbildung 4.1: Verteilung der ermittelten *Total Tile Length* aus Experiment 3.

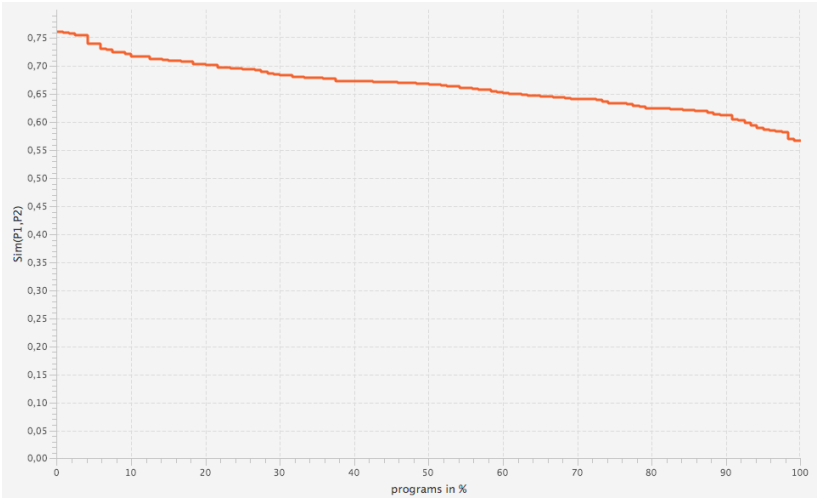


Abbildung 4.2: Verteilung der ermittelten *Sim(p1,p2)*-Werte aus Experiment 3.

4.4 Experiment 4: Vergleich mit einem anderen Plagiatserkennungsprogramm

In diesem Experiment werden die Ergebnisse von *PF* mit denen von *JPlag*, einem Plagiaterkennungsprogramm der Universität Karlsruhe, verglichen. *JPlag* verwendet genauso wie *PF* den *GST* [MP00, S. 12]. Es wird nicht ein normalisierter Java-Bytecode, sondern ein geparster Quellcode verglichen. Der Parsvorgang übersetzt Strukturen des Quellcodes in Bezeichnungen und hängt diese in einer Zeichenkette aneinander. Zum Beispiel wird einer Zeichenkette für den Anfang einer Klasse *„beginclass“* und für das Ende *„endclass“* hinzugefügt [MP00, S 10-11]. Da *JPlag* die Struktur eines Programms vergleicht ist es ein *SMS* (s. Abs. 2.1). Es wird im Folgenden angenommen, dass *JPlags* Maß *„mittlere Häufigkeit“* dem Maß $Sim(p1, p2)$ entspricht. Dies kann nicht bestätigt werden, da sich die Hilfe von *JPlag* [Mal] und das Paper über *JPlag* [MP00] in der Definition des Maßes widersprechen. Tests lassen darauf schließen, dass das Maß $Sim(p1, p2)$ verwendet worden ist. Ein versuchter E-Mail-Kontakt mit den Entwicklern von *JPlag* lieferte kein Ergebnis. Näheres zu *JPlag* findet sich in [MP00].

4.4.1 Vorgehensweise

Zuerst wird ein Benutzerkonto zur Verwendung von *JPlag* angefordert. Danach wird der *JPlag*-Client heruntergeladen, mit dessen Hilfe auf den *JPlag*-Server zugegriffen werden kann. Es werden mit *JPlag* alle von Dritten erstellten Daten aus Experiment 2 (Abs. 4.2) auf Plagiate untersucht und mit den Ergebnissen von *Plagiarism Finder* verglichen. Das selbe Vorgehen wird mit den Originalabgaben aus Experiment 3 (Abs. 4.3) umgesetzt. Die *MML* wird auf den gleichen Wert wie in jenen Experimenten gesetzt. Die Quellcodedateien müssen aus den „.jar“-Dateien extrahiert werden, da *JPlag* nur „.java“-Dateien, aber keine „.jar“-Dateien vergleichen kann. Programme können verglichen werden, indem zu jedem Programm ein Ordner angelegt wird, welcher den Quellordner inklusive aller Pakete und Quellcodedateien des Programms enthält. Bei der Untersuchung von Dritten erstellten Programme wird überprüft, ob Programme falsch-positiv oder falsch-negativ als Plagiate erkannt werden. Ein Programm wird falsch-negativ erkannt, wenn sich der Ähnlichkeitswert durch eine plagiierter Änderung verringert. Ein Programm wird als falsch-positiv erkannt, wenn der Ähnlichkeitswert bei einer nicht-plagiierten Änderung unverändert bleibt. Bei der Untersuchung studentischer Abgaben wird die Abweichung der gegenübergestellten $Sim(p1, p2)$-Werte untersucht.

4.4.2 Ergebnis

Die Pars- und Vergleichszeit für den Originaldatensatz studentischer Abgaben beträgt 90,1 Sekunden. Die Auflistung aller Ergebnisdaten im Vergleich mit den ermittelten Daten von *PF* findet sich in C.5 und C.6. Tabelle 4.4 beinhaltet einen Auszug der Gegenüberstellung der von *PF* und *JPlag* ermittelten Ähnlichkeitswerte bei der Untersuchung von Dritten erstellter Programme. Man kann erkennen, dass sowohl falsch-positiv als auch falsch-negativ erkannte Plagiate existieren. Tabelle 4.5 zeigt einen Ausschnitt der ermittelten Ähnlichkeitswerte aus der Untersuchung der studentischen Abgaben. Es ist zu erkennen, dass die gegenübergestellten $Sim(p1, p2)$-Werte eine geringe Abweichung haben. Die höchste Abweichung liegt bei 0.116 die niedrigste bei 0.

Programm 1 (p1)	PF Sim(p1,p2)	JPlag Sim(p1,p2)	Stufe	Änderung
S1.jar	1	1,000	1	JavaDoc entfernt
S3.jar	1	0,913	3	Änderung der Reihenfolge von Methoden, Datenfelder und Konstruktoren
S5.jar	0,978	1,000	3	Alle Zugriffsmodifikatoren auf „package-private“ geändert
S9.jar	0,995	1,000	nicht plagiiert	„+=“ in „-=“ geändert
S10.jar	0,978	0,965	5	While statt For-Schleife
S12.jar	1,000	0,991	4	„super()“ Aufrufe entfernt

Tabelle 4.4: Gegenüberstellung der ermittelten *Sim(p1,p2)*-Werte von *PF* und *JPlag* von der Untersuchung der von Dritten erstellten Programme. Die Spalte *„Stufen“* bedeutet, dass eine Änderung auf dieser Stufe vorgenommen worden ist. Sie bedeutet nicht, dass alle Änderungen niedrigerer Stufen vorhanden sind. Die Stufe *nicht plagiiert* bedeutet, dass eine nicht-plagiierte Änderung vorgenommen worden ist. *Programm 2 (p2)* ist immer das Programm *Original.jar*. Rote Zelle: *Programm 1* ist falsch-negativ als Plagiat erkannt worden. Grüne Zelle: *Programm 1* ist falsch.positiv als Plagiat erkannt worden.

Programm 1 (p1)	Programm 2 (p2)	PF Sim(p1,p2)	JPlag Sim(p1,p2)	Abweichung
331.jar	338.jar	0,755	0,871	0,116
363.jar	365.jar	0,616	0,646	0,030
336.jar	358.jar	0,644	0,659	0,015
321.jar	354.jar	0,729	0,733	0,004
302.jar	360.jar	0,722	0,721	0,001
360.jar	365.jar	0,709	0,709	0,000

Tabelle 4.5: Auszug aus der Gegenüberstellung der ermittelten *Sim(p1,p2)*-Werte von *PF* und *JPlag* bei der Untersuchung studentischer Abgaben.

5 Diskussion und Ausblick

Dieses Kapitel beinhaltet die Diskussion der Experimentergebnisse (Abs.5.1), eine weiterführende Diskussion über *PF* (Abs. 5.2) und einen Ausblick über die Weiterentwicklung von *PF* sowie über zukünftige Forschungsthemen (Abs. 5.3).

5.1 Interpretation und Diskussion der Experimentergebnisse

In diesem Abschnitt wird auf die Ergebnisse der Experimente aus Abschnitt 4 eingegangen. Diese werden ausgewertet, interpretiert und diskutiert. Zusätzlich werden Bezüge zu *PF* hergestellt.

5.1.1 Diskussion von Experiment 1 und 2

Die Ergebnisse aus Experiment 1 und 2 zeigen, dass Änderungen auf der Stufe 1 und 2 keinen Einfluss auf die *TTL* haben. Die Stufe 3 hat in einem von 4 Fällen, die Stufe 4 in 5 von 10 Fällen, die Stufe 5 in allen Fällen und die Stufe 6 in elf von vierzehn Fällen einen Einfluss auf die *TTL*. Die Änderungen der *TTL* in Stufe 3 folgen daraus, dass durch die Änderung von Zugriffsmodifikatoren im Quellcode im Bytecode durch andere JVM-Instruktionen auf Datenfelder zugegriffen wird. Die Änderungen in Stufe 4 entstehen, weil durch das Auslagern von Code in Submethoden zusätzliche Befehle zum Aufruf der Submethode in die aufrufende Methode eingefügt werden. Weiterhin führt das Entfernen von *„static“* Schlüsselwörtern zu zusätzlichen *„putfield“* und *„aload“* Befehlen. For- und Whileschleifen werden durch verschiedene Befehlssequenzen abgebildet. Daher folgen die Änderungen in Stufe 5. Da in Stufe 6 die Logik des Codes verändert wird, werden andere Befehle verwendet. Dies führt wiederum zu einer Änderung der *TTL*.
PF wird nach diesem Ergebnis in die Stufe 2 eingestuft. Da die *TTL* durch Änderungen auf den Stufen 1 und 2 stabil bleibt. *PF* erkennt allerdings auch viele Änderungen höherer Stufen.

5.1.2 Diskussion von Experiment 3

In Experiment 3 werden keine eindeutigen Plagiate erkannt. Es existieren $Sim(p)$-Werte, welche mit 0.8 deutlich über dem Mittelwert von 0.67 liegen. Ausschlaggebend hierfür ist, dass häufig ein Quellcode, welcher eine gleiche Struktur aber eine andere Bedeutung als das Original hat, als plagiiert erkannt wird. Zum Beispiel ist ein Code zum Erstellen mehrerer Ameisenobjekten mit einem Code zum Erstellen mehrerer Futterhaufen- oder Ameisenhaufenobjekten gleich gesetzt worden. Zusätzlich werden häufig Teile der Konstruktoren als plagiiert erkannt, da die Befehlscodes der Bytecodebefehle zum Laden verschiedener Datenfelder meist gleich sind.
Hier wird deutlich, dass zum Erkennen von Plagiaten nicht nur die Struktur eines Programms und die semantische Bedeutung der Struktur betrachtet werden kann. Wortlaute und Namen sind durchaus notwendig um den Kontext eines Codes verstehen zu können. Zum Beispiel ist es ein Unterschied, ob man durch eine Iteration 50 Ameisen oder 50 Ameisenhaufen erstellt. Durch Analyse der Struktur kann dieser Unterschied nicht erkannt werden, da von Namen abstrahiert wird und so 50 Instanzen von irgendeiner Klasse erstellt werden. Um dies zu erkennen, muss ein Weg gefunden werden, wie ein Plagiatserkennungsprogramm erkennen kann, ob zwei Typen mit unterschiedlichen Typbezeichnungen den gleichen Typ darstellen.
Obwohl keine Plagiate erkannt worden sind, kann der Ansatz zur Erkennung von Vorlagen (Abs. 2.3) zumindest teilweise validiert werden. Wie in Abbildung 4.1 zu sehen, liegt der Grenzwert bei 2528. Man könnte diesen als den Wert für die Größe der Vorlage wählen. Besser geeignet ist der Median, da durch ihn Änderungen an der Vorlage ignoriert werden können. Dieser beträgt hier 2610. Viele der niedrigeren *TTLs* ergeben sich durch Änderungen an der Vorlage. Höhere *TTLs* sind auf kurze sowie unbedeutende Übereinstimmungen des von den Studierenden programmierten Codes, sowie auf fälschlicherweise erkannte Übereinstimmungen zurückzuführen. Durch dieses Experiment wird gezeigt, dass Vorlagen durch die dargestellte Verteilung der *TTL* erkannt werden können.
Ändert man den verwendeten Datensatz so ab, dass Programme doppelt vorhanden sind, werden *TTLs* erkannt, welche deutlich über dem Median liegen. Zum Beispiel ergibt der Vergleich von „302.jar“ mit sich selbst eine *TTL* von 3964. Die Differenz zum Median beträgt 1354. Hiermit wird gezeigt, dass der Ansatz zur Erkennung von Vorlagen anwendbar ist.

5.1.3 Diskussion von Experiment 4

Es ist aufgrund der unbekannten Länge des von *JPlag* geparsten Codes nicht möglich, die *TTL* aus den von *JPlag* ermittelten Daten zu bestimmen. Deshalb wird der Wert $Sim(p1, p2)$ betrachtet. Dieser ist zwar abhängig von der Gesamtlänge des Programms, allerdings haben die Änderungen, welche zur Erstellung der Programme vorgenommen worden sind, nur selten und wenn, dann einen sehr kleinen Einfluss auf die Länge der

Programme.

Diskussion des Vergleichs der von Dritten erstellten Plagiate

JPlag wird genauso wie in *PF* nach den Stufen des Plagiierens (Abs.2.6) in Stufe 2 eingestuft, da es durch Änderung auf Stufe 1 und 2 zu keinem Einfluss auf den *Sim(p1,p2)*-Wert und somit auf die *TTL* kommt. Bei höheren Stufen haben die plagiierten Änderungen einen Einfluss. Besonders anzumerken ist hier, dass das Verschieben von Methoden in *Plagiarism Finder* keinen Einfluss auf die Ähnlichkeit hat, in *JPlag* allerdings schon. Die Sensitivität von *PF* beträgt 0,43 und bei *JPlag* 0,53. Die Spezifität von PF ist 1 und bei *JPlag* 0,5. Bei der Bestimmung der Sensitivität werden alle plagiierten Änderungen, bei welchen $Sim(p1, p2)$ konstant bleibt, als positiv erkannt gezählt. Als tatsächlich positiv gelten alle Daten mit plagiierten Änderungen. Bei Bestimmung der Spezifität werden alle nicht-plagiierten Änderungen, bei welchen sich $Sim(p1, p2)$ ändert, als negativ erkannt gezählt. Als tatsächlich negativ gelten alle Daten mit nicht-plagiierten Änderungen. Da sich *PF*, durch Ignorieren kleiner Methoden, wie in Abschnitt 3.3.2 beschrieben, bei der Bestimmung von $Sim(p1, p2)$ anders verhält als *JPlag*, können keine genaueren analytischen Werte bestimmt werden.
Es zeigt sich, dass der in Abschnitt 3.3.2 beschriebene Ansatz zur Vermeidung von Änderung der Ähnlichkeitswerte bei Verschieben von Methoden an diesen Beispieldaten funktioniert. Bei *JPlag* ändern sich diese Werte. Negativ zu bewerten ist, dass letzteres nur an einer Beispieldatei getestet worden ist. Es zeigt sich anhand der Spezifität, dass *PF* deutlich besser im Einordnen von nicht-plagiierten Änderungen ist. Vergleicht man die Spezifität, ist *JPlag* erfolgreicher im Einordnen von plagiierten Änderungen. Die berechneten Werte für die Sensitivität und Spezifität sowie die daraus hergeleitete Folgerung gelten nur für den hier verwendeten Datensatz. Es kann nicht auf die Allgemeinheit geschlossen werden, da der Datensatz, aufgrund fehlender Gleichverteilung der Änderungsstufen nicht repräsentativ ist.

Diskussion des Vergleichs der studentischen Abgaben

Die Bearbeitungszeit zum Parsen und Vergleichen der studentischen Abgaben hat in *JPlag* 1.5 Minuten, in *Plagiarism Finder* 8.98 Minuten gedauert. Man könnte daraus schließen, dass *JPlag* schneller ist, allerdings ist nichts über die Hardware des *JPlag*-Servers bekannt.
Obwohl sich die $Sim(p1, p2)$-Werte der beiden Plagiatserkennungsprogramme aufgrund der unterschiedlich definierten Gesamtlänge nicht exakt vergleichen lassen, erhält man trotzdem ähnliche Ergebnisse. Die durchschnittliche Abweichung der ermittelten Werte liegt bei $0,0159$. Diese ist über $\frac{\sum |a-b|}{n}$ bestimmt worden, wobei a und b die jeweiligen Sim-Werte darstellen und n die Gesamtmenge an Werten. Dass die Abweichung hier gering ist, obwohl sich Sensitivität und Spezifität der Programme

unterscheiden, hat zur Ursache, dass in den Abgaben wenige plagiierten Änderungen der Stufe 2 oder höher vorhanden sind. Der Großteil der ermittelten Ähnlichkeit zweier Programme beruht auf der unveränderten Kopie der Vorlage sowie auf wenige plagiierten Änderungen der ersten und zweiten Stufe. Durch die Tatsache, dass beide Programme im Erkennen dieser Änderungen gleich gut sind (Abs. 5.1.1,5.1.3) und durch den *GST* alle direkten Kopien erkannt werden, lassen sich die geringen Abweichungen erklären.
Dies zeigt, dass sich die Ergebnisse von *JPlag* und *PF* bei wörtlichen Plagiaten und intelligenten Plagiaten auf niedriger Stufe kaum unterscheiden. Da dies die häufigsten Arten des Plagiierens sind, können in den meisten Fällen beide Programme gleichberechtigt verwendet werden.

5.2 Diskussion der entwickelten Plagiatserkennungssoftware

Dieser Abschnitt beinhaltet eine Diskussion über *PF*. Einige Aspekte über *PF* sind bereits in Abschnitt 5.1 erwähnt worden. Auch wenn *PF* im Erkennen von Plagiaten mit *JPlag* mithalten kann, hat er doch einige Schwachstellen. Alle zu vergleichenden Programme müssen mit dem gleichen Compiler kompiliert werden, da verschiedene Compiler manche Quellcodestrukturen anders übersetzen. Dies würde die Ähnlichkeit beeinflussen. Das heißt, um einen Vergleich möglichst effektiv durchführen zu können, muss jedes zu vergleichende Programm zuerst mit einem identischen Compiler kompiliert werden. Dazu muss der Quellcode vorhanden sein. Da das Compilieren auch ein Parsen beinhaltet, erhält man in dieser Hinsicht keinen Vorteil durch Verwendung eines normalisierten Bytecodes anstellte eines geparsten Quellcodes.
Es existieren JVM-Instruktionen, welche ihre Referenzen in ihrem Befehlscode und nicht in den Operanden beinhalten. Zum Beispiel besitzt der Befehlscode 47 (dstore_0) keine Operanden. Dennoch speichert er einen Double-Wert in die erste lokale Variable der Variablentabelle. Solche JVM-Instruktionen werden in *PF* nicht besonders behandelt, obwohl es die Absicht war, alle Referenzen zu entfernen. Da nur durch das Verändern der Reihenfolge der Erzeugung von lokalen Variablen aus einem *„dstore_ 0“* ein *„dstore_ 1“* werden kann, führt die Missachtung dieser Befehlscodes zu ungenaueren Ergebnissen.
Dass die Berechnung der Ergebnisse von Experiment 2 „7,79“ Minuten gedauert hat, stellt die Nützlichkeit von *PF* im Realbetrieb in Frage. Es ist hier anzumerken, dass das Augenmerk bei der Entwicklung von *PF* nicht auf der Performance lag. Trotz der Parallelisierung der Vergleiche, ist die gesamte Vergleichszeit hoch, da die einzelnen Vergleiche lange dauern. Dies könnte durch Implementierung einer schnelleren Version des *GST* gelöst werde.
Des Weiteren steht der Praxistauglichkeit von *PF* die fehlende Unterstützung von Java 8 im Wege. Es kann nur Quellcode verglichen werden, welcher keine Java 8 spezifischen

Elemente enthält. Die Unterstützung von Java 8 ist nicht umgesetzt worden, da *BCEL* in ihrem aktuellen Stand nicht in der Lage ist, Java 8 Bytecode zu verarbeiten.
Ein weiter wichtiger Punkt ist, dass *PF* zwar alle Programme miteinander vergleicht, sich aber immer nur die Überschneidungen zweier Programme anzeigen lassen. Es wäre eine Ansicht wünschenswert, in welcher zu einem Programm alle Übereinstimmungen mit jeglichen anderen Programmen angezeigt werden kann. Da der *GST* nicht unbedingt optimal ist (s. Abs. 2.2.3), kann es sein, dass nicht alle Übereinstimmungen gefunden werden. Des Weiteren ist nicht überprüft worden, in wie weit es möglich ist, dass verschiedenen Quellcodes zum gleichen Bytecode führen. Ist dies möglich, würden solche Quellcodes von *PF* falsch-positiv als Plagiate erkannt werden.
Positiv anzumerken ist, dass alle in Abschnitt 3.1 gesetzten Anforderungen an *PF* sowie die am Anfang dieser Arbeit aufgeführten Ziele (Abs. 1.2) erfüllt worden sind.

5.3 Ausblick

Obwohl *PF* gut zum Erkennen von Plagiaten geeignet ist, kann an ihm einiges verbessert werden. Zum einem ist die Performance mangelhaft und muss erhöht werden. Hierzu ist die Implementierung einer schnelleren Version des *GST* vonnöten. Um mit *PF* Java 8 Quellcode unterstützen zu können, ist es geplant, *BCEL* durch eine andere Programmierschnittstelle zur Bytecodeverarbeitung, welche Java 8 Klassendateien unterstützt, zu ersetzen. Zusätzlich wird eine Ansicht implementiert, in welcher zu einem Programm alle Übereinstimmungen mit jeglichen anderen Programmen angezeigt werden kann. Außerdem werden einige Fehler, welche beim Öffnen der Quellcodeansicht auftreten, behoben. Des Weiteren ist es geplant eine Überprüfung von Sensibilität und Spezifität anhand einer repräsentativen Stichprobe durchzuführen.
Zukünftige Arbeit im wissenschaftlichen Sinne besteht im Finden eines Weges, wie Plagiatserkennungsprogramme die Bedeutung eines Quellcodes besser verstehen können. Dies kann beispielsweise durch Interpretation des Wortlautes erreicht werden. Ein erster Schritt in diese Richtung wäre es, zu lösen, wie ein Plagiatserkennungsprogramm erkennen kann, dass zwei Typen mit unterschiedlichen Typbezeichnungen den gleichen Typ darstellen. Zudem muss analysiert werden, in wie weit es möglich ist, dass das Kompilieren verschiedener Quellcodes in einem gleichen Bytecode resultiert. Ein letzter wichtiger Aspekt wäre eine Entwicklung eines sowohl optimalen als auch effizienten Vergleichsalgorithmus.

Literaturverzeichnis

[17] *Duden, Das Große Fremdwörterbuch.* Dudenverlag, 2000.

[ASA12] Salah M. Alzahrani, Naomie Salim und Ajith Abraham. "Understanding Plagiarism Linguisitc Patterns, Textual Features, and Detection Methods". In: *Systems, Man, and Cybernetics* 42.2 (03/2012).

[Blo08] Joshua Bloch. *Effective Java.* 2. Aufl. Addison-Wesley, 05/2008.

[BU98] Brenda S. Beka und Manber Udi. "Deducing Similarities in Java Sources from Bytecodes". In: *Proceedings of the USENIX Annual Technical Conference* 98 (1998), S. 15.

[Clo00] Paul Clough. *Plagiarism in natural and programming languages: an overview of current tools and technologies.* Techn. Ber. Western Bank Sheffield S10 2TN UK: Department of Computer Science, University of Sheffield, 07/2000.

[Fou] Apache Software Foundation. *Apache Commons BCEL.* URL: `http://commons.apache.org/proper/commons-bcel/index.html` (besucht am 08. 07. 2014).

[Gra02] Andrew Granville. "Detecting Plagiarism in Java Code". Bachelorarbeit. Western Bank Sheffield S10 2TN UK: University of Sheffield, 05/2002.

[Hal77] Maurive Howard Halstead. *Elements of software science.* Elsevier, 1977.

[Koi] Sami Koivu. *reJ.* URL: `http://rejava.sourceforge.net/index.html` (besucht am 25. 06. 2014).

[Lin+11] Tim Lindholm, Frank Yellin, Gilad Bracha und Alex Buckley. *The Java® Virtual Machine Specification Java SE 7 Edition.* 500 Oracle Parkway Redwood City California 94065 U.S.A.: Oracle America, 07/2011.

[Mal] Guido Malphol. *JPlag Match ranking.* URL: `https://jplag.ipd.kit.edu/example/help-sim-en.html` (besucht am 08. 07. 2014).

[MP00] Lutz Perchelt und Guido Malpohl und Michael Philippsen. *JPlag: Finding plagiarisms among a set of programs.* Techn. Ber. 1. D-76128 Karlsruhe, Germany: Universität Karlsruhe, 03/2000.

[Mül07] Klaus Müller. "Vergleich und Implementierung von Verfahren zur Plagiaterkennung von Software". Abschlussarbeit. Fachhochschule Trier, 09/2007.

[Ott76] K.J. Ottstein. "An algorithmic approach to the detection and prevention of plagiarism". In: *ACM SIGSCE Bulletin* 8.4 (1976), S. 30–41.

[SKR86] J. A. W. Faidhi und S. K. Robinson. "An emperical approach for detecting programm similarity and plagiarism within a university programming environment". In: *Computers a. Education* 11.1 (03/1986), S. 11–19.

[Wes95] Adrian West. *Coping with plagiarism in Computer Science teaching laboratories.* Techn. Ber. The Victoria University of Manchester Oxford RoadManchester M13 9P: Victoria University of Manchester, 07/1995.

[Wis94] Michael J. Wise. *String Similarity via Greedy String Tiling and Running KarpRabin Matching.* Techn. Ber. University of Sydney, F09 N. S. W. 2006 Australia: Department of Computer Science, University of Sydney, 1994.

[Wis96] Michael J. Wise. *YAP3: improved detection of similarities in computer program and other texts.* Techn. Ber. Department of Computer Science, University of Sydney, 1996.

[WV96] Michael J. Wise und Krisitna L. Verco. *Software for Detecting Suspected Plagiarism: Comparing Structure and Attribute-Counting Systems.* Techn. Ber. The University of Sydney NSW 2006 Australia: Department of Computer Science University of Sidney, 1996.

Appendix

A Klassendiagramm von Plagiarism Finder

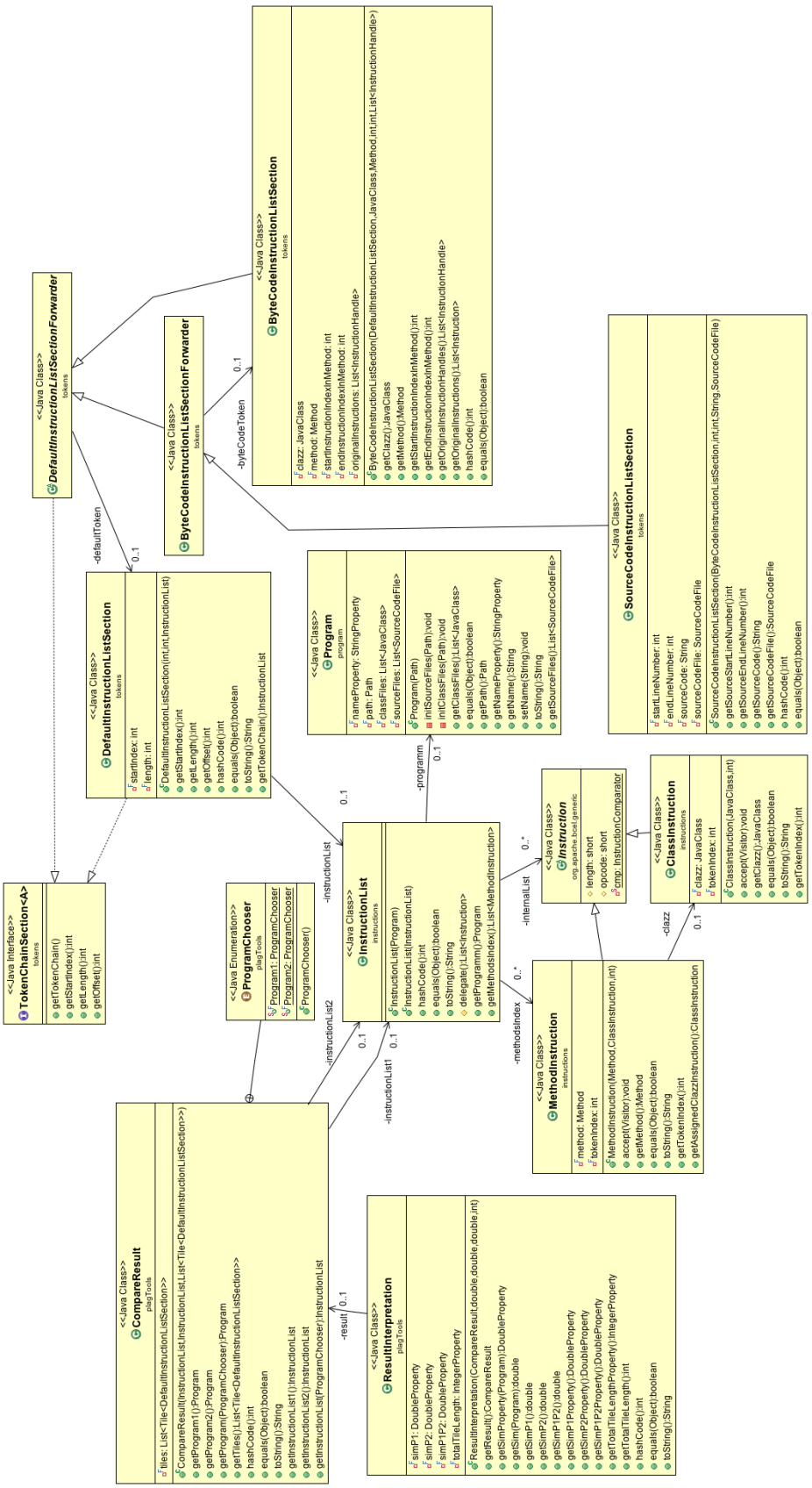


Abbildung A.1: Klassendiagramm der Modellklassen von *Plagiarism Finder*.

B Anleitung zur Plagiatserstellung

Anleitung zur Erstellung von Plagiaten und Nicht-Plagiaten

Allgemeines:

- Erstelle einige Plagiate und Nicht-Plagiate.
- Bemühe dich um eine angemessene Wortwahl bei Änderungen.
- Verwende keinen Java 8 Code.

Vorgehen:

1. Importiere das original Projekt und verändere es-
2. Verändere immer nur eine logische Einheit (z.B. Methodennamen).
3. Dokumentiere extern (nicht im Code!) was du verändert hast. Tue dies in der Form:

Projektname	Plagiat	Änderung
Klassenname	ja	Klassennamen A in B umbenannt

4. Exportiere uns Sende das veränderte Projekt als .jar Datei mit Sourcecode oder sende direkt die .java-Dateien.

Erstellen von Plagiaten:

- Die Logik, also der Befehlsfluss bzw. die Reihenfolge der Befehle darf sich nicht ändern.
- Mögliche Änderungskritieren:
  - Klassen- oder Attribut- oder lokale Variablennamen
  - Klassen- oder Methoden oder Attributreihenfolge
  - Sichtbarkeit von Methoden, Klassen, Attributen
  - Einfügen oder Löschen von Sprachbausteinen (final ...)
  - Kommentare ändern
  - Zusätzliche oder weniger Importe
  - Verschiedene Execeptions , also statt "throws Exception" , eine konkrete Exception (z.B. NullpointerException) angeben
  - Zusätzliche Annotationen
  - Code in private Methoden auslagern
  - Zusätzliche Methoden bzw. Klassen einfügen, welche den ursprünglichen Befehlsfluss nicht verändern
  - Kleine Änderungen in der Logik mit gleichem Ergebnis. Z.B. a+=1 statt a = a+1
  - ...

Erstellen von Nicht-Plagiaten:

- Die Logik, also der Befehlsfluss bzw. die Reihenfolge der Befehle muss sich ändern.
- Interessant sind hier
  - kleine Änderungen im Code, welche zu einem komplett anderen Verhalten des Programms führen.
  - große Änderungen die zum gleichen Verhalten des Programms führen
  - komplett andere Lösungsansätze zu verwenden, um auf das gleiche Verhalten zu kommen.
  - ...

Abbildung B.1: Die in Experiment 2 verwendete Anleitung zum Erstellen von Plagiaten. Ein Plagiat oder Nicht-Plagiat bezieht sich in diesem Kontext auf eine plagiierte oder nicht-plagiierte Änderung.

C Ergebnistabellen der Experimente

C.1 Ergebnisdaten Experiment 1

Programm1	Programm2	sim(p1)	sim(p2)	sim(p1,p2)	TTL	Stufe
ChangedAttributeNames.jar	Original.jar	1	1	1	96	2
ChangedAttributeOrders.jar	Original.jar	1	1	1	96	3
ChangedClassNames.jar	Original.jar	1	1	1	96	2
ChangedClassPosition.jar	Original.jar	1	1	1	96	3
ChangedComments.jar	Original.jar	1	1	1	96	1
ChangedIdentations.jar	Original.jar	1	1	1	96	1
ChangedImports.jar	Original.jar	1	1	1	96	3
ChangedLogic.jar	Original.jar	0,8868	0,9792	0,9307	94	6
ChangedMethodOrder.jar	Original.jar	1	1	1	96	3
ChangedNewAttributes.jar	Original.jar	1	1	1	96	4
ChangedNewMethods.jar	Original.jar	0,8496	1	0,9187	96	4
ChangedParameterNames.jar	Original.jar	1	1	1	96	2
ChangedRemoveFinals.jar	Original.jar	1	1	1	96	4
ChangedVisibilities.jar	Original.jar	1	1	1	96	4
ChangedWhiletoFor.jar	Original.jar	0,787	0,8854	0,8333	85	5
ChangeLocalVarNames.jar	Original.jar	1	1	1	96	2

Tabelle C.2: Ergebnis des Vergleichs eigener Beispieldaten mit *PF* bei einer *MML* von 4. Der Inhalt der Spalte *"Stufe"* bedeutet, dass eine Änderung auf dieser Stufe vorgenommen wurde. Er bedeutet nicht, dass alle Änderungen niedrigerer Stufen vorhanden sind.

C.2 Ergebnisdaten Experiment 2

Programm1 (p1)	Sim(p1)	Sim(p2)	Sim(p1,p2)	TTL	Änderung	Stufe
S1.jar	1	1	1	181	JavaDoc entfernt	1
S2.jar	1	1	1	181	Neuer JavaDoc eingefügt	1
S3.jar	1	1	1	181	Reihenfolge von Methoden, Datenfelder und Konstruktoren geändert	3
S5.jar	0.9779	0.9779	0.9779	177	Alle Zugriffsmodifikatoren auf „package-private“ geändert	3
S6.jar	1	1	1	181	Zusätzliche Importe	4
S7.jar	1	1	1	181	Exceptiontyp geändert („NullPointerException“ zu „IllegalArgumentException“)	6
S8.jar	0.9031	0.9779	0.9390	177	Code in Submethoden ausgelagert	4
S9.jar	0.9945	0.9945	0.9945	180	„+=“ in „-=“	nicht plagiiert
S10.jar	0.9779	0.9779	0.9779	177	FOR- in WHILE-Schleife	5
S11.jar	0.6029	0.9227	0.7293	167	Zusätzliche Textausgaben	6
S12.jar	1	1	1	181	„super()“ Aufrufe entfernt	4
S13.jar	0.7864	0.9558	0.8628	173	Zusätzliches Datenfeld und Manipulation dieses Datenfeldes	6
S14.jar	0.8894	0.9779	0.9316	177	Explizierte Initialisierung von Datenfeldern	6

Programm1 (p1)	Sim(p1)	Sim(p2)	Sim(p1,p2)	TTL	Änderung	Stufe
S15.jar	1	1	1	181	Entfernung überflüssiger Klammern	1
S16.jar	1	1	1	181	Entfernung von Whitespaces	1
S17.jar	0.9676	0.9890	0.9781	179	„Math.pow“ durch „x*x“ ersetzt	6
S18.jar	1	1	1	181	„x+=y“ durch „x = x+y“ ersetzt	6
S19.jar	0.9779	0.9779	0.9779	177	„runlist.size() == 0“ durch „runlist.isEmty()“ ersetzt	6
S20.jar	1	1	1	181	Autorennamen in Javadoc geändert	1
S21.jar	0.9382	0.9227	0.9304	167	Code aus aufgerufener Methode direkt in die aufrufende Methoden eingesetzt	4
S22.jar	0.8840	0.8840	0.8840	160	double Cast entfernt. „Double/int“ zu „int/int“	nicht plagiiert
S23.jar	0.9679	1	0.9837	181	Zusätzliche Exception geworfen	nicht plagiiert
R1.jar	0.9890	0.9945	0.9917	180	Rückgabewert von „int“ auf „long“ gändert	6
R2.jar	0.8956	0.9006	0.8981	163	FOR-EACH- in FOR-Schleife	5
R3.jar	0.9676	0.9890	0.9781	179	„Math.pow“ durch „x*x“ ersetzt	6
R4.jar	0.9511	0.9669	0.9589	175	„public static class“ zu „static class“	4
R5.jar	1	1	1	181	Entfernen aller „final“ Schlüsselworte	4
R6.jar	1	1	1	181	Umbenennung einer Klasse	2
R7.jar	0.9227	0.9227	0.9227	167	List in Arraylist	6

Programm1 (p1)	Sim(p1)	Sim(p2)	Sim(p1,p2)	TTL	Änderung	Stufe
R8.jar	0.9945	0.9945	0.9945	180	„+=“ in „-=“	nicht plagiiert
R9.jar	0.9945	0.9945	0.9945	180	$>$ in $<$	nicht plagiiert
G1.jar	0.9521	0.9890	0.9702	179	Klassenname geändert , Kommentare hinzugefügt und neue Exception geworfen	6
G2.jar	0.2250	0.1492	0.1794	27	Komplette Neuimplementierung der Logik	nicht plagiiert
G3.jar	0.9521	0.9890	0.9702	179	Zusätzliche und andere Exceptions	6
G4.jar	1	1	1	181	Umbenennung von Klassen, Methoden, Datenfeldern, lokalen Variablen	2
G5.jar	0.9779	0.9779	0.9779	177	Alle Zugriffsmodifikatoren auf „protected“ geändert	4

Tabelle C.3: Ergebnis des Vergleichs von Dritten erstellten Beispieldaten bei einer *MML* von 4.Der Inhalt der Spalte *„Stufe“* bedeutet, dass eine Änderung auf dieser Stufe vorgenommen worden ist. Er bedeutet nicht, dass alle Änderungen niedrigerer Stufen vorhanden sind. *„p2“* ist immer *„Original.jar“*. Die Stufe *„nicht plagiiert“* bedeutet, dass eine nicht-plagiierte Änderung vorgenommen wurde. Die Datei „S4“ konnte nicht kompiliert werden und ist deshalb nicht aufgelistet.

C.3 Ergebnisdaten Experiment 3

Programm1 (p1)	Programm2 (p2)	Sim(p1)	Sim(p2)	Sim(p1,p2)	TTL
353.jar	360.jar	0,5692	0,7805	0,6583	2905
323.jar	358.jar	0,6591	0,6245	0,6413	2844
353.jar	358.jar	0,5531	0,6199	0,5846	2823
336.jar	358.jar	0,6712	0,6186	0,6438	2817
338.jar	353.jar	0,7659	0,5505	0,6406	2810
311.jar	358.jar	0,7287	0,6164	0,6679	2807
311.jar	360.jar	0,7277	0,7531	0,7402	2803
338.jar	360.jar	0,7632	0,7523	0,7577	2800
323.jar	353.jar	0,6487	0,5484	0,5943	2799
336.jar	360.jar	0,6640	0,7488	0,7039	2787
311.jar	353.jar	0,7233	0,5458	0,6222	2786
302.jar	353.jar	0,7028	0,5458	0,6145	2786
311.jar	323.jar	0,7222	0,6447	0,6813	2782
358.jar	360.jar	0,6107	0,7472	0,6721	2781
323.jar	360.jar	0,6429	0,7453	0,6903	2774
302.jar	360.jar	0,6995	0,7450	0,7216	2773
302.jar	323.jar	0,6985	0,6417	0,6689	2769
319.jar	353.jar	0,6845	0,5386	0,6029	2749
336.jar	353.jar	0,6540	0,5378	0,5903	2745
311.jar	331.jar	0,7116	0,8050	0,7554	2741
302.jar	358.jar	0,6897	0,6004	0,6419	2734
311.jar	338.jar	0,7082	0,7435	0,7254	2728
360.jar	363.jar	0,7324	0,6337	0,6795	2726
302.jar	311.jar	0,6864	0,7064	0,6963	2721
323.jar	363.jar	0,6290	0,6309	0,6299	2714
338.jar	358.jar	0,7394	0,5957	0,6599	2713
360.jar	365.jar	0,7284	0,6914	0,7094	2711
331.jar	360.jar	0,7959	0,7281	0,7605	2710
338.jar	363.jar	0,7375	0,6290	0,6790	2706
302.jar	338.jar	0,6819	0,7367	0,7082	2703
309.jar	360.jar	0,6836	0,7262	0,7043	2703
309.jar	323.jar	0,6798	0,6229	0,6501	2688
323.jar	338.jar	0,6227	0,7324	0,6731	2687

Programm1 (p1)	Programm2 (p2)	Sim(p1)	Sim(p2)	Sim(p1,p2)	TTL
319.jar	360.jar	0,6683	0,7211	0,6937	2684
353.jar	363.jar	0,5255	0,6234	0,5703	2682
302.jar	363.jar	0,6756	0,6225	0,6480	2678
358.jar	363.jar	0,5878	0,6223	0,6046	2677
311.jar	336.jar	0,6934	0,6364	0,6637	2671
331.jar	338.jar	0,7841	0,7277	0,7549	2670
331.jar	358.jar	0,7841	0,5863	0,6709	2670
302.jar	309.jar	0,6728	0,6745	0,6737	2667
302.jar	336.jar	0,6723	0,6350	0,6531	2665
336.jar	338.jar	0,6345	0,7258	0,6771	2663
323.jar	336.jar	0,6160	0,6333	0,6245	2658
336.jar	363.jar	0,6328	0,6174	0,6250	2656
309.jar	358.jar	0,6712	0,5828	0,6239	2654
309.jar	365.jar	0,6702	0,6758	0,6730	2650
323.jar	354.jar	0,6139	0,7218	0,6635	2649
302.jar	354.jar	0,6680	0,7215	0,6937	2648
309.jar	353.jar	0,6692	0,5184	0,5842	2646
311.jar	363.jar	0,6867	0,6148	0,6488	2645
319.jar	323.jar	0,6579	0,6123	0,6343	2642
354.jar	358.jar	0,7193	0,5797	0,6420	2640
354.jar	365.jar	0,7185	0,6725	0,6948	2637
331.jar	353.jar	0,7744	0,5167	0,6198	2637
358.jar	365.jar	0,5788	0,6723	0,6221	2636
319.jar	363.jar	0,6559	0,6123	0,6333	2634
311.jar	319.jar	0,6833	0,6554	0,6690	2632
319.jar	331.jar	0,6549	0,7724	0,7088	2630
302.jar	331.jar	0,6627	0,7715	0,7130	2627
353.jar	365.jar	0,5145	0,6697	0,5819	2626
319.jar	358.jar	0,6536	0,5764	0,6126	2625
309.jar	311.jar	0,6636	0,6812	0,6723	2624
331.jar	354.jar	0,7686	0,7131	0,7398	2617
309.jar	331.jar	0,6616	0,7683	0,7110	2616
311.jar	354.jar	0,6789	0,7125	0,6953	2615
323.jar	331.jar	0,6060	0,7680	0,6775	2615
331.jar	365.jar	0,7674	0,6664	0,7133	2613

Programm1 (p1)	Programm2 (p2)	Sim(p1)	Sim(p2)	Sim(p1,p2)	TTL
336.jar	365.jar	0,6223	0,6662	0,6435	2612
311.jar	365.jar	0,6776	0,6656	0,6716	2610
350.jar	354.jar	0,6850	0,7098	0,6972	2605
311.jar	321.jar	0,6755	0,7646	0,7173	2602
321.jar	365.jar	0,7643	0,6634	0,7103	2601
319.jar	365.jar	0,6477	0,6634	0,6554	2601
323.jar	365.jar	0,6028	0,6634	0,6316	2601
302.jar	319.jar	0,6557	0,6472	0,6514	2599
354.jar	360.jar	0,7076	0,6977	0,7027	2597
302.jar	365.jar	0,6549	0,6621	0,6585	2596
309.jar	338.jar	0,6560	0,7070	0,6806	2594
350.jar	358.jar	0,6821	0,5696	0,6208	2594
321.jar	331.jar	0,7620	0,7615	0,7618	2593
302.jar	350.jar	0,6541	0,6818	0,6677	2593
309.jar	336.jar	0,6558	0,6178	0,6362	2593
319.jar	338.jar	0,6447	0,7056	0,6738	2589
321.jar	338.jar	0,7602	0,7051	0,7316	2587
331.jar	350.jar	0,7589	0,6795	0,7170	2584
321.jar	350.jar	0,7590	0,6792	0,7169	2583
321.jar	360.jar	0,7582	0,6932	0,7242	2580
331.jar	336.jar	0,7577	0,6147	0,6788	2580
338.jar	354.jar	0,7029	0,7027	0,7028	2579
321.jar	354.jar	0,7576	0,7025	0,7290	2578
319.jar	354.jar	0,6414	0,7019	0,6703	2576
311.jar	350.jar	0,6685	0,6771	0,6728	2575
309.jar	319.jar	0,6507	0,6407	0,6457	2573
302.jar	321.jar	0,6488	0,7558	0,6982	2572
350.jar	360.jar	0,6763	0,6910	0,6836	2572
350.jar	365.jar	0,6763	0,6560	0,6660	2572
353.jar	354.jar	0,5039	0,7008	0,5863	2572
309.jar	363.jar	0,6502	0,5976	0,6228	2571
321.jar	358.jar	0,7546	0,5639	0,6455	2568
338.jar	350.jar	0,6988	0,6742	0,6863	2564
319.jar	336.jar	0,6382	0,6107	0,6241	2563
309.jar	354.jar	0,6477	0,6978	0,6718	2561

Programm1 (p1)	Programm2 (p2)	Sim(p1)	Sim(p2)	Sim(p1,p2)	TTL
336.jar	354.jar	0,6092	0,6967	0,6501	2557
338.jar	365.jar	0,6966	0,6519	0,6735	2556
321.jar	323.jar	0,7496	0,5912	0,6611	2551
331.jar	363.jar	0,7489	0,5927	0,6617	2550
323.jar	350.jar	0,5910	0,6705	0,6282	2550
321.jar	353.jar	0,7488	0,4992	0,5990	2548
321.jar	336.jar	0,7485	0,6069	0,6703	2547
319.jar	321.jar	0,6315	0,7452	0,6837	2536
363.jar	365.jar	0,5890	0,6463	0,6163	2534
350.jar	353.jar	0,6647	0,4953	0,5676	2528
354.jar	363.jar	0,6883	0,5872	0,6337	2526
309.jar	321.jar	0,6378	0,7411	0,6856	2522
319.jar	350.jar	0,6250	0,6600	0,6420	2510
309.jar	350.jar	0,6325	0,6576	0,6448	2501
336.jar	350.jar	0,5959	0,6576	0,6253	2501
321.jar	363.jar	0,7323	0,5793	0,6469	2492
350.jar	363.jar	0,6526	0,5769	0,6125	2482

Tabelle C.4: Ergebnis des Vergleichs von studentischen Abgaben bei einer *MML* von 10. Die Tabelle ist absteigend nach der *TTL* sortiert. Die Datei „344.jar“ ist nicht aufgelistet, da sie nicht entpackt werden konnte.

C.4 Ergebnisdaten Experiment 4

Programm1 (p1)	PF Sim(p1,p2)	JPlag Sim(p1,p2)	Stufe	Änderung
S1.jar	1	1,000	1	JavaDoc entfernt
S2.jar	1	1,000	1	Neuer JavaDoc eingefügt
S3.jar	1	0,913	3	Reihenfolge von Methoden, Datenfelder und Konstruktoren geändert
S5.jar	0,978	1,000	3	Alle Zugriffsmodifikatoren auf „package-private“ geändert
S6.jar	1,000	1,000	4	Zusätzliche Importe
S7.jar	1,000	1,000	6	Exceptiontyp geändert („NullPointerException“ zu „IllegalArgumentException“)
S8.jar	0,939	0,867	4	Code in Submethoden ausgelagert
S9.jar	0,995	1,000	nicht plagiiert	„+=“ in „-=“
S10.jar	0,978	0,965	5	FOR- in WHILE-Schleife
S11.jar	0,729	0,679	6	Zusätzliche Textausgaben
S12.jar	1,000	0,991	4	„super()“ Aufrufe entfernt
S13.jar	0,863	0,991	6	Zusätzliches Datenfeld und Manipulation dieses Datenfeldes
S14.jar	0,932	0,941	6	Explizierte Initialisierung von Datenfeldern
S15.jar	1,000	1,000	1	Entfernung überflüssiger Klammern
S16.jar	1,000	1,000	1	Entfernung von Whitespaces
S17.jar	0,978	0,995	6	„Math.pow“ durch „x*x“ ersetzt
S18.jar	1,000	1,000	6	„x+=y“ durch „x = x+y“ ersetzt
S19.jar	0,978	1,000	6	„runlist.size() == 0“ durch „runlist.isEmty()“ ersetzt
S20.jar	1,000	1,000	1	Autorennamen in Javadoc geändert
S21.jar	0,930	0,982	4	Code aus aufgerufener Methode direkt in die aufrufende Methoden eingesetzt
S22.jar	0,884	1,000	nicht plagiiert	double Cast entfernt. Double / int zu int / int
S23.jar	0,984	0,983	nicht plagiiert	Zusätzliche Exception geworfen

Programm1 (p1)	PF Sim(p1,p2)	JPlag Sim(p1,p2)	Stufe	Änderung
R1.jar	0,992	1,000	6	Rückgabewert von „int“ auf „long“ geändert
R2.jar	0,898	0,974	5	FOR-EACH- in FOR-Schleife
R3.jar	0,978	0,995	6	„Math.pow“ durch „x*x“ ersetzt
R4.jar	0,959	1,000	4	„public static class“ zu „static class“
R5.jar	1,000	1,000	4	Entfernen aller final Schlüsselworte
R6.jar	1,000	1,000	2	Umbenennung einer Klasse
R7.jar	0,923	0,995	6	List in Arraylist
R8.jar	0,995	0,995	nicht plagiiert	„+=“ in „-=“
R9.jar	0,995	1,000	nicht plagiiert	> in <
G1.jar	0,970	0,983	6	Klassenname geändert, Kommentare hinzugefügt, neue Exception geworfen
G2.jar	0,179	0,763	nicht plagiiert	Komplette Neuimplementierung der Logik
G3.jar	0,970	0,983	6	Zusätzliche und andere Exceptions
G4.jar	1,000	1,000	2	Umbenennung von Klassen, Methoden, Datenfeldern, Lokalen Variablen
G5.jar	0,978	1,000	4	Alle Zugriffsmodifikatoren auf „protected geändert“

Tabelle C.5: Gegenüberstellung der ermittelten $Sim(p1, p2)$-Werte zwischen *PF* und *JPlag* bei Untersuchung der von Dritten erstellten Daten. Die *MML* ist 4. Der Inhalt der Spalte *„Stufen“* bedeutet, dass eine Änderung auf dieser Stufe vorgenommen wurde. Sie bedeutet nicht, dass alle Änderungen niedrigerer Stufen vorhanden sind. Die Stufe *nicht plagiiert* bedeutet, dass eine nicht-plagiierte Änderung vorgenommen worden ist. *„p2“* ist immer *„Original.jar“*. Rote Zelle: Wert verringert sich bei plagiierter Änderung. Programm ist falsch-negativ als Plagiat erkannt worden. Grüne Zelle: Wert bleibt bei nicht plagiierter Änderung konstant. Programm wird falsch-positiv als Plagiat erkannt.

Programm1 (p1)	Programm2 (p2)	PF Sim(p1,p2)	Sim(p1,p2)	Abweichung
331.jar	338.jar	0,755	0,871	0,116
323.jar	331.jar	0,677	0,769	0,092
323.jar	338.jar	0,673	0,743	0,070
338.jar	360.jar	0,758	0,698	0,060
321.jar	365.jar	0,710	0,752	0,042
311.jar	331.jar	0,755	0,714	0,041
350.jar	358.jar	0,621	0,654	0,033
338.jar	365.jar	0,674	0,705	0,031
350.jar	353.jar	0,568	0,599	0,031
323.jar	360.jar	0,690	0,659	0,031
321.jar	353.jar	0,599	0,63	0,031
319.jar	363.jar	0,633	0,603	0,030
350.jar	365.jar	0,666	0,696	0,030
363.jar	365.jar	0,616	0,646	0,030
358.jar	360.jar	0,672	0,698	0,026
336.jar	353.jar	0,590	0,616	0,026
331.jar	336.jar	0,679	0,704	0,025
302.jar	323.jar	0,669	0,644	0,025
353.jar	354.jar	0,586	0,611	0,025
338.jar	353.jar	0,641	0,616	0,025
353.jar	358.jar	0,585	0,609	0,024
309.jar	323.jar	0,650	0,626	0,024
331.jar	365.jar	0,713	0,737	0,024
353.jar	360.jar	0,658	0,635	0,023
319.jar	353.jar	0,603	0,58	0,023
311.jar	365.jar	0,672	0,694	0,022
311.jar	338.jar	0,725	0,703	0,022
309.jar	319.jar	0,646	0,668	0,022
336.jar	350.jar	0,625	0,647	0,022
321.jar	358.jar	0,645	0,667	0,022
323.jar	363.jar	0,630	0,61	0,020
319.jar	336.jar	0,624	0,644	0,020
319.jar	331.jar	0,709	0,689	0,020
321.jar	360.jar	0,724	0,743	0,019

Programm1 (p1)	Programm2 (p2)	PF Sim(p1,p2)	Sim(p1,p2)	Abweichung
309.jar	336.jar	0,636	0,655	0,019
319.jar	360.jar	0,694	0,675	0,019
319.jar	354.jar	0,670	0,689	0,019
302.jar	338.jar	0,708	0,69	0,018
302.jar	363.jar	0,648	0,63	0,018
358.jar	365.jar	0,622	0,64	0,018
358.jar	363.jar	0,605	0,622	0,017
311.jar	350.jar	0,673	0,69	0,017
338.jar	358.jar	0,660	0,677	0,017
338.jar	363.jar	0,679	0,662	0,017
311.jar	363.jar	0,649	0,632	0,017
302.jar	321.jar	0,698	0,715	0,017
338.jar	350.jar	0,686	0,703	0,017
321.jar	336.jar	0,670	0,686	0,016
319.jar	358.jar	0,613	0,628	0,015
319.jar	321.jar	0,684	0,699	0,015
336.jar	358.jar	0,644	0,659	0,015
302.jar	311.jar	0,696	0,682	0,014
311.jar	360.jar	0,740	0,726	0,014
309.jar	321.jar	0,686	0,699	0,013
354.jar	363.jar	0,634	0,647	0,013
311.jar	323.jar	0,681	0,668	0,013
302.jar	354.jar	0,694	0,707	0,013
309.jar	358.jar	0,624	0,637	0,013
331.jar	358.jar	0,671	0,684	0,013
353.jar	365.jar	0,582	0,595	0,013
311.jar	319.jar	0,669	0,682	0,013
311.jar	358.jar	0,668	0,655	0,013
302.jar	365.jar	0,658	0,671	0,013
331.jar	360.jar	0,760	0,773	0,013
323.jar	358.jar	0,641	0,629	0,012
331.jar	353.jar	0,620	0,632	0,012
336.jar	363.jar	0,625	0,637	0,012
321.jar	338.jar	0,732	0,743	0,011

Programm1 (p1)	Programm2 (p2)	PF Sim(p1,p2)	Sim(p1,p2)	Abweichung
353.jar	363.jar	0,570	0,581	0,011
302.jar	309.jar	0,674	0,663	0,011
323.jar	336.jar	0,625	0,614	0,011
309.jar	311.jar	0,672	0,662	0,010
323.jar	354.jar	0,663	0,673	0,010
309.jar	360.jar	0,704	0,695	0,009
309.jar	350.jar	0,645	0,654	0,009
321.jar	350.jar	0,717	0,726	0,009
323.jar	350.jar	0,628	0,637	0,009
311.jar	321.jar	0,717	0,726	0,009
350.jar	360.jar	0,684	0,692	0,008
319.jar	338.jar	0,674	0,682	0,008
319.jar	350.jar	0,642	0,65	0,008
302.jar	350.jar	0,668	0,675	0,007
311.jar	354.jar	0,695	0,688	0,007
321.jar	331.jar	0,762	0,769	0,007
302.jar	331.jar	0,713	0,72	0,007
354.jar	358.jar	0,642	0,649	0,007
309.jar	331.jar	0,711	0,704	0,007
309.jar	363.jar	0,623	0,616	0,007
302.jar	319.jar	0,651	0,658	0,007
323.jar	353.jar	0,594	0,588	0,006
331.jar	354.jar	0,740	0,746	0,006
309.jar	354.jar	0,672	0,666	0,006
321.jar	323.jar	0,661	0,666	0,005
311.jar	353.jar	0,622	0,627	0,005
338.jar	354.jar	0,703	0,698	0,005
350.jar	363.jar	0,612	0,617	0,005
360.jar	363.jar	0,679	0,675	0,004
321.jar	363.jar	0,647	0,651	0,004
321.jar	354.jar	0,729	0,733	0,004
354.jar	365.jar	0,695	0,691	0,004
336.jar	338.jar	0,677	0,674	0,003
336.jar	354.jar	0,650	0,647	0,003

Programm1 (p1)	Programm2 (p2)	PF Sim(p1,p2)	Sim(p1,p2)	Abweichung
331.jar	350.jar	0,717	0,72	0,003
309.jar	365.jar	0,673	0,676	0,003
319.jar	323.jar	0,634	0,637	0,003
354.jar	360.jar	0,703	0,7	0,003
319.jar	365.jar	0,655	0,658	0,003
336.jar	365.jar	0,644	0,646	0,002
302.jar	353.jar	0,614	0,612	0,002
309.jar	338.jar	0,681	0,683	0,002
331.jar	363.jar	0,662	0,664	0,002
350.jar	354.jar	0,697	0,695	0,002
302.jar	358.jar	0,642	0,64	0,002
311.jar	336.jar	0,664	0,662	0,002
323.jar	365.jar	0,632	0,63	0,002
302.jar	336.jar	0,653	0,654	0,001
336.jar	360.jar	0,704	0,703	0,001
309.jar	353.jar	0,584	0,585	0,001
302.jar	360.jar	0,722	0,721	0,001
360.jar	365.jar	0,709	0,709	0,000

Tabelle C.6: Gegenüberstellung der ermittelten *Sim(p1,p2)*-Werte von *PF* und *JPlag* bei der Untersuchung studentischer Abgaben. Die *MML* ist 10. Die Tabelle ist absteigend nach der Abweichung sortiert.

D Digitales Begleitmaterial

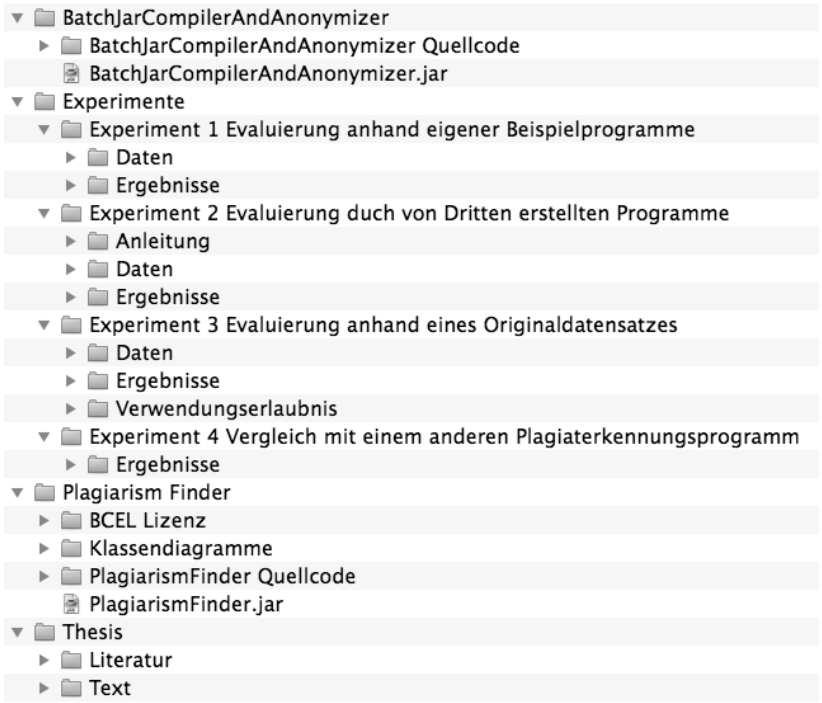


Abbildung D.1: Ordnerstruktur der beiliegenden CD mit dem digitalen Begleitmaterial.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen (einschließlich elektronischer Quellen) direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Heilbronn, 21. Juli 2014

_____

(Unterschrift)